Biodata Penulis

Marti Widya Sari lahir di Yogyakarta pada tanggal 27 Maret 1979. Beliau merupakan dosen pada Program Studi Informatika, Fakultas Sains dan Teknologi, Universitas PGRI Yogyakarta. Latar belakang pendidikannya dimulai dari Sarjana (S1) dan Magister (S2) di Departemen Teknik Elektro dan Teknologi Informasi, Fakultas Teknik, Universitas Gadjah Mada (UGM). Selanjutnya, beliau menempuh pendidikan Doktor (S3) di Program Studi Teknik Industri, Fakultas Teknik, Universitas Gadjah Mada. Kepakaran beliau berfokus pada bidang Internet of Things (IoT), dan transformasi digital di era Industri 4.0. Beliau aktif melakukan penelitian dan publikasi ilmiah baik nasional maupun internasional yang berorientasi pada pengembangan sistem berbasis IoT. Selain kegiatan penelitian, beliau juga terlibat dalam berbagai program pengabdian kepada masyarakat, khususnya dalam penerapan teknologi tepat guna dan inovasi digital. Dengan semangat pengembangan ilmu pengetahuan dan inovasi, beliau berkomitmen untuk terus berkontribusi dalam pengembangan dunia pendidikan, riset, dan industri melalui inovasi digital yang berkelanjutan dan berdampak nyata bagi pembangunan bangsa.

UPY Press
Lembaga Penelitian dan Pengabdian Masyarakat Unit 1 Gedung B Lantai 2
Jl. PGRI I Sonosewu No. 117 Yogyakarta
Telp (0274) 376808, 373198,418077, Fax (0274) 376808
Email: upypress@gmail.com, Web: upypress.upy.ac.id

ISBN 978-623-8551-40-8
9 786238 551408

Dr. Marti Widya Sari, S.T., M.Eng

Sistem Operasi Modern

UPY Press

Sistem Operasi Modern

Penulis :
Dr. Marti Widya Sari, S.T., M.Eng

2025

SISTEM OPERASI MODERN

Penulis :

Dr. Marti Widya Sari, S.T., M.Eng

UPY Press

SISTEM OPERASI MODERN

Penulis : Dr. Marti Widya Sari, S.T., M.Eng
Editor : Reza Diapratama, S.Kom
Layout : Prayitno
Cover : Reza Diapratama, S.Kom

Cetakan Pertama, Oktober 2025
17 cm x 23 cm + v + 106

ISBN : 978-623-8551-40-8

Penerbit :
UPY Press
Lembaga Penelitian dan Pengabdian Masyarakat
Unit 4 Lantai 4
Jl. PGRI I Sonosewu No. 117 Yogyakarta
Telp (0274) 376808, 373198,418077, Fax (0274) 376808
Email: upypress@gmail.com
Web: upypress.upy.ac.id

Hak cipta dilindungi oleh Undang-Undang
Dilarang memperbanyak karya tulisan ini tanpa izin tertulis dari Penerbit

Cetakan I, Oktober 2025

KATA PENGANTAR

Segala puji syukur penulis panjatkan ke hadirat Allah SWT atas limpahan rahmat dan karunia-Nya sehingga buku dengan judul "Sistem Operasi Modern" ini dapat disusun dan diselesaikan dengan baik.

Buku ini disusun sebagai salah satu bahan ajar untuk mendukung kegiatan pembelajaran dalam mata kuliah Sistem Operasi, khususnya yang membahas perkembangan sistem operasi modern. Materi dalam buku ini mencakup berbagai topik penting mulai dari arsitektur sistem operasi, virtualisasi, cloud computing, sistem embedded, hingga keamanan dan integrasi kecerdasan buatan. Harapannya, buku ini dapat menjadi panduan bagi mahasiswa dan pembaca umum untuk memahami esensi dan arah perkembangan sistem operasi di era komputasi masa kini.

Penyusunan buku ini juga didasarkan pada berbagai referensi terkini dan studi literatur dari sumber-sumber akademik yang kredibel, serta dilengkapi dengan contoh-contoh nyata dari sistem operasi modern yang digunakan secara luas saat ini, seperti Windows 11, macOS, Linux, Android, dan iOS.

Penulis menyadari masih terdapat kekurangan dalam buku ini. Oleh karena itu, kritik dan saran yang membangun sangat diharapkan demi penyempurnaan edisi selanjutnya.

Semoga buku ini dapat memberikan manfaat yang sebesar-besarnya bagi seluruh pembaca dan menjadi bagian dari upaya peningkatan mutu pendidikan di bidang teknologi informasi.

Yogyakarta, 30 September 2025

Penulis

DAFTAR ISI

BAB 1: PENDAHULUAN SISTEM OPERASI MODERN

Sistem operasi (SO) merupakan fondasi utama yang memungkinkan perangkat keras komputer berfungsi dan berinteraksi dengan perangkat lunak aplikasi. Seiring dengan perkembangan teknologi informasi yang pesat, sistem operasi juga terus berevolusi dari bentuk tradisionalnya menjadi sistem operasi modern yang memiliki kapabilitas lebih canggih dan adaptif terhadap berbagai lingkungan komputasi. Evolusi ini didorong oleh kebutuhan akan performa yang lebih tinggi, keamanan yang lebih baik, efisiensi sumber daya, serta kemampuan untuk mendukung paradigma komputasi baru seperti *cloud computing*, *mobile*, dan *Internet of Things* (IoT). Bab ini akan menguraikan definisi sistem operasi modern, membandingkannya secara fundamental dengan sistem operasi tradisional, membahas tren dan inovasi terkini yang membentuk lanskap SO saat ini, serta memberikan contoh sistem operasi modern yang dominan dan banyak digunakan di berbagai platform.

A. Definisi Sistem Operasi Modern

Sistem Operasi (SO) secara fundamental adalah perangkat lunak sistem yang berfungsi sebagai manajer sumber daya perangkat keras komputer dan perangkat lunak, sekaligus menyediakan layanan dasar untuk program aplikasi. Ini termasuk manajemen prosesor, memori, perangkat I/O (input/output), serta sistem *file*. SO bertindak sebagai antarmuka perantara, menyederhanakan interaksi kompleks antara pengguna, aplikasi, dan perangkat keras, sehingga pengguna dapat

menjalankan program tanpa perlu memahami detail teknis tingkat rendah dari perangkat keras.

Namun, "sistem operasi modern" memiliki konotasi yang lebih spesifik, mencerminkan evolusi signifikan dari pendahulunya. SO modern adalah sistem yang dirancang untuk mengatasi tantangan dan memanfaatkan peluang dari arsitektur komputasi kontemporer dan kebutuhan pengguna yang semakin kompleks. Karakteristik utama yang mendefinisikan sistem operasi modern mencakup:

1. Dukungan Penuh untuk Arsitektur Multi-core dan Paralelisme Berbeda dengan SO tradisional yang mungkin hanya mendukung satu inti prosesor atau *multi-tasking* sederhana, SO modern dioptimalkan untuk memanfaatkan sepenuhnya arsitektur *multi-core* dan *multi-threaded*. Ini melibatkan algoritma penjadwalan proses yang canggih untuk mendistribusikan beban kerja secara efisien ke berbagai inti CPU, memungkinkan eksekusi paralel dari banyak tugas dan meningkatkan responsivitas sistem secara keseluruhan.
2. Arsitektur Modular dan Mikrokernel (atau Hybrid) Sebagian besar SO tradisional mengadopsi arsitektur monolitik, di mana semua komponen inti (manajemen memori, *device driver*, sistem *file*, dll.) berada dalam satu blok kernel besar. SO modern seringkali beralih ke arsitektur yang lebih modular, termasuk desain mikrokernel (kernel minimal dengan layanan inti di ruang pengguna) atau *hybrid kernel*. Pendekatan ini meningkatkan stabilitas (karena kegagalan satu komponen di ruang pengguna tidak meruntuhkan seluruh kernel), keamanan (isolasi antar komponen), dan kemudahan pemeliharaan serta pembaruan.
3. Manajemen Memori Virtual yang Canggih SO modern mengimplementasikan sistem memori virtual yang robust, memungkinkan aplikasi untuk mengakses memori lebih dari yang

tersedia secara fisik. Ini dicapai melalui teknik *paging* dan *swapping*, serta alokasi memori dinamis. Selain itu, perlindungan memori antarproses sangat krusial, mencegah satu aplikasi untuk mengakses atau merusak ruang memori aplikasi lain atau kernel itu sendiri, yang meningkatkan stabilitas dan keamanan sistem.

4. Mekanisme Keamanan Terintegrasi yang Kuat Dengan semakin maraknya ancaman siber, keamanan menjadi prioritas utama. SO modern memiliki lapisan keamanan berlapis yang terintegrasi, termasuk kontrol akses wajib (Mandatory Access Control/MAC) seperti SELinux dan AppArmor, yang menerapkan kebijakan keamanan ketat di luar kendali pengguna. Fitur *sandboxing* dan isolasi proses memastikan bahwa aplikasi berbahaya tidak dapat mempengaruhi bagian lain dari sistem. Enkripsi *native* (misalnya, enkripsi *full-disk*) juga menjadi standar untuk melindungi data saat *rest*.
5. Dukungan Jaringan yang Komprehensif dan Terdistribusi SO modern dibangun dengan kemampuan jaringan yang mendalam, mendukung berbagai protokol komunikasi (TCP/IP, UDP, dll.) dan layanan jaringan yang kompleks. Ini memungkinkan konektivitas yang lancar ke internet, *cloud services*, dan arsitektur terdistribusi, di mana sumber daya dan komputasi tersebar di beberapa mesin.
6. Antarmuka Pengguna Grafis (GUI) yang Intuitif dan Responsif Evolusi dari *Command Line Interface* (CLI) ke GUI telah membuat komputer lebih mudah diakses. SO modern menawarkan GUI yang kaya fitur visual, mendukung interaksi multi-sentuh, suara, bahkan *gesture*. Desain yang responsif memastikan pengalaman pengguna yang lancar di berbagai ukuran layar dan perangkat.

7. Skalabilitas dan Fleksibilitas SO modern dirancang untuk beradaptasi dengan berbagai skala dan jenis perangkat keras, mulai dari perangkat *embedded* berdaya rendah (misalnya, untuk IoT), *smartphone*, tablet, laptop, desktop, hingga server *enterprise* dan infrastruktur *cloud* yang masif. Fleksibilitas ini memungkinkan SO yang sama atau turunannya untuk digunakan dalam berbagai konteks komputasi.
8. Dukungan Virtualisasi dan Kontainerisasi Kemampuan untuk menjalankan beberapa sistem operasi atau aplikasi terisolasi pada satu perangkat keras fisik adalah ciri khas SO modern. SO modern menyediakan dukungan bawaan untuk teknologi *hypervisor* (seperti Type 1 dan Type 2) untuk virtualisasi mesin virtual (VM) dan juga mendukung kontainerisasi (misalnya, Docker) untuk isolasi aplikasi yang lebih ringan dan efisien.

B. Perbandingan Sistem Operasi Tradisional dan Modern

Untuk memahami sepenuhnya esensi sistem operasi modern, penting untuk melihat perbedaannya dengan sistem operasi tradisional yang mendahuluinya. Perbedaan ini tidak hanya terletak pada fitur permukaan, tetapi juga pada filosofi desain, arsitektur internal, dan respons terhadap kebutuhan komputasi pada masanya.

1. Sistem Operasi Tradisional (Contoh: MS-DOS, Awal Windows 95, Unix Awal)
2. Fokus Operasional: Umumnya dirancang untuk komputasi *single-user* atau *single-tasking*, atau *batch processing* pada era *mainframe*. Interaksi *real-time* dengan banyak aplikasi secara bersamaan masih terbatas.
3. Arsitektur Kernel: Mayoritas menggunakan arsitektur monolitik. Seluruh komponen kernel (penjadwal CPU, manajemen memori,

device driver, sistem *file*, layanan jaringan) dikompilasi menjadi satu blok kode besar yang berjalan di *privileged mode*. Keuntungan dari arsitektur ini adalah performa yang cepat karena semua komponen berada dalam ruang alamat yang sama. Namun, kelemahannya adalah stabilitas yang rentan – *bug* atau *driver* perangkat yang tidak stabil dapat menyebabkan *kernel panic* dan meruntuhkan seluruh sistem.

4. Antarmuka Pengguna: Umumnya berbasis Command Line Interface (CLI) seperti MS-DOS, di mana pengguna mengetikkan perintah teks. Beberapa memiliki GUI sangat dasar (misalnya, Windows 1.0 atau awal Windows 95) yang terbatas fungsionalitasnya.
5. Manajemen Sumber Daya: Terbatas. Manajemen memori seringkali sederhana tanpa perlindungan memori yang kuat antarprogram. Penjadwalan proses primitif, kurang efisien untuk *multi-tasking* yang berat.
6. Jaringan: Kapabilitas jaringan sangat minim atau tidak ada sama sekali secara *native*. Implementasi jaringan biasanya memerlukan penambahan *software* pihak ketiga yang kompleks.
7. Keamanan: Fokus pada perlindungan *file* dasar dan otentikasi *single-user*. Mekanisme keamanan tidak dirancang untuk menahan ancaman siber kompleks atau lingkungan *multi-user* yang hostile. Konsep *privileged mode* dan *user mode* ada, tetapi isolasi kurang ketat.
8. Skalabilitas: Rendah. Sulit untuk diadaptasi ke perangkat keras baru atau diperluas untuk mendukung skala komputasi yang lebih besar.
9. Virtualisasi/Kontainerisasi: Sama sekali tidak mendukung konsep ini secara *native*, karena virtualisasi memerlukan abstraksi perangkat keras yang canggih yang tidak dimiliki SO tradisional.

10. Sistem Operasi Modern (Contoh: Windows 11, macOS, Linux, Android, iOS)
11. Fokus Operasional: Dirancang untuk komputasi *multi-tasking*, *multi-user*, *client-server*, komputasi terdistribusi, *cloud computing*, dan *mobile*. Mampu menjalankan banyak aplikasi secara bersamaan dengan responsivitas tinggi.
12. Arsitektur Kernel: Cenderung mengadopsi arsitektur modular, mikrokernel, atau *hybrid kernel*.
    a. Mikrokernel: Kernel inti sangat kecil, hanya menangani fungsi dasar seperti manajemen memori tingkat rendah, penjadwalan proses, dan inter-process communication (IPC). Sebagian besar layanan SO (sistem *file*, *device driver*, jaringan) berjalan sebagai proses terpisah di ruang pengguna (misalnya, QNX). Ini meningkatkan stabilitas dan keamanan.
    b. Hybrid Kernel: Menggabungkan elemen monolitik dan mikrokernel, di mana beberapa layanan non-esensial dipindahkan ke ruang pengguna, tetapi *device driver* dan layanan penting lainnya tetap di kernel untuk performa (misalnya, Windows NT Family, macOS).
    c. Modular Monolitik: Kernel Linux secara teknis adalah monolitik, tetapi sangat modular dengan kemampuan memuat dan membongkar modul *driver* secara dinamis, memberikan fleksibilitas tanpa mengorbankan performa.
    d. Antarmuka Pengguna: Dominan Antarmuka Pengguna Grafis (GUI) yang canggih, intuitif, responsif, dan kaya fitur visual. Mendukung berbagai *input* (sentuhan, suara, *gesture*, *stylus*) dan adaptif terhadap resolusi layar yang berbeda.
    e. Manajemen Sumber Daya: Sangat canggih. Menggunakan *preemptive multi-tasking* untuk memastikan setiap proses mendapat waktu CPU yang adil. Manajemen memori virtual

dengan *demand paging* dan *swapping* yang efisien. Sistem *file* yang *journaling* untuk integritas data.

f. Jaringan: Terintegrasi penuh dengan berbagai protokol jaringan dan API untuk *cloud services*, VPN, *firewall*, dan komunikasi *peer-to-peer*. Dirancang untuk lingkungan yang selalu terhubung.
g. Keamanan: Dibangun dengan filosofi "keamanan berlapis" (defense-in-depth). Ini mencakup:
    1) Mandatory Access Control (MAC): Kebijakan akses yang ditentukan oleh sistem administrator, bukan pengguna, untuk keamanan yang lebih tinggi.
    2) Isolasi Proses dan Sandboxing: Setiap aplikasi berjalan dalam lingkungan terisolasi untuk mencegah akses tidak sah atau kerusakan sistem.
    3) Enkripsi Data: Fitur enkripsi *full-disk* atau *file-level* secara bawaan untuk melindungi data dari akses fisik tidak sah.
    4) Pembaruan Otomatis: Mekanisme *patch management* dan *update* keamanan reguler untuk menambal kerentanan.
h. Skalabilitas: Tinggi, dapat menyesuaikan performa dan fitur dari perangkat berdaya rendah (IoT) hingga server *enterprise* bertenaga tinggi, serta lingkungan *cloud* berskala *petabyte*.
i. Virtualisasi/Kontainerisasi: Dukungan bawaan untuk teknologi *hypervisor* (baik Type 1 maupun Type 2) dan *container runtime* (misalnya, Docker). Ini memungkinkan efisiensi sumber daya dan isolasi aplikasi yang lebih baik, menjadi fundamental dalam arsitektur *cloud* dan *microservices*.

C. Tren dan Inovasi Teknologi Sistem Operasi

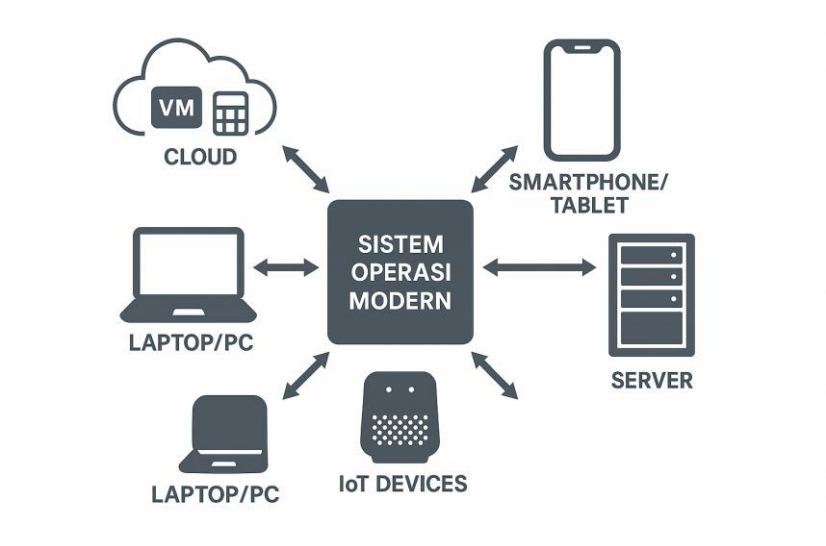
Lanskap teknologi yang terus berubah menuntut sistem operasi untuk terus berinovasi. Beberapa tren dan inovasi kunci yang secara signifikan membentuk arah pengembangan sistem operasi modern meliputi:

1. Komputasi Awan (Cloud Computing) dan Virtualisasi:
   a. SO sebagai *Guest* di *Cloud*: Banyak SO modern dioptimalkan untuk berjalan sebagai mesin virtual (VM) di infrastruktur *cloud* publik maupun privat. Ini memerlukan kernel yang efisien dalam lingkungan virtual dan *driver* yang kompatibel dengan *hypervisor*.
   b. SO sebagai *Host* untuk *Cloud*: SO seperti Linux banyak digunakan sebagai sistem operasi *host* untuk *hypervisor* (misalnya, KVM) yang menjalankan VM di *cloud*.
   c. Abstraksi Sumber Daya: Inovasi dalam SO *cloud* berfokus pada abstraksi sumber daya fisik menjadi sumber daya virtual yang dapat disekalakan secara dinamis, memungkinkan model *Infrastructure as a Service* (IaaS) dan *Platform as a Service* (PaaS).
2. Internet of Things (IoT) dan Edge Computing:
   a. SO Ringan dan Hemat Daya: Perangkat IoT memiliki sumber daya yang sangat terbatas (memori, prosesor, daya). Ini mendorong pengembangan SO yang sangat ringan dan *real-time* seperti FreeRTOS, Zephyr OS, dan RIOT OS, yang dirancang untuk footprint kecil, konsumsi daya minimal, dan kemampuan respons cepat untuk aplikasi kritis.
   b. Keamanan di Edge: Keamanan menjadi tantangan besar di perangkat IoT. SO untuk *edge computing* berinovasi dalam fitur keamanan perangkat keras (Hardware Root of Trust),

firmware update yang aman, dan isolasi proses untuk melindungi data di tepi jaringan.

3. Integrasi Kecerdasan Buatan (AI) dan Pembelajaran Mesin (ML):
   a. Optimasi Sumber Daya Adaptif: AI/ML digunakan dalam SO untuk mempelajari pola penggunaan pengguna dan aplikasi, kemudian mengoptimalkan alokasi sumber daya (CPU, memori, I/O) secara dinamis untuk performa yang lebih baik dan efisiensi energi.
   b. Peningkatan Keamanan Prediktif: ML dapat menganalisis perilaku sistem dan mendeteksi anomali yang mengindikasikan serangan siber atau *malware* lebih cepat daripada metode deteksi berbasis tanda tangan tradisional.
   c. Antarmuka Pengguna Cerdas: Asisten virtual (misalnya, Cortana, Siri, Google Assistant) semakin terintegrasi dalam SO, menggunakan AI untuk memahami perintah suara dan memberikan respons kontekstual.
4. Kontainerisasi dan *Microservices*:
   a. Dukungan Kernel untuk Kontainer: SO modern, terutama Linux, menyediakan fitur kernel (seperti *namespaces* dan *cgroups*) yang menjadi dasar bagi teknologi kontainer seperti Docker dan Kubernetes. Ini memungkinkan aplikasi untuk dikemas dan dijalankan dalam lingkungan terisolasi yang lebih ringan dan cepat daripada VM.
   b. Arsitektur *Microservices*: Kontainer memfasilitasi adopsi arsitektur *microservices*, di mana aplikasi dipecah menjadi layanan-layanan kecil yang independen. SO modern harus efisien dalam mengelola banyak kontainer yang berjalan secara bersamaan.

5. Keamanan Proaktif dan Privasi Data:
    a. Zero-Trust Security: SO modern bergerak menuju model keamanan *zero-trust*, di mana setiap permintaan akses diverifikasi tanpa asumsi kepercayaan.
    b. Perlindungan Data *End-to-End*: Peningkatan dalam enkripsi data di seluruh tumpukan, dari penyimpanan (data at rest) hingga transmisi (data in transit).
    c. Privasi Pengguna: SO kini memberikan kontrol yang lebih granular kepada pengguna atas izin aplikasi dan akses data pribadi, sejalan dengan regulasi privasi seperti GDPR.
6. Pengembangan Open Source dan Kolaborasi Komunitas:
    a. Model pengembangan *open source*, yang dipelopori oleh Linux, telah membuktikan diri sebagai pendorong inovasi yang kuat. Komunitas global dapat meninjau kode, menemukan *bug*, mengembangkan fitur baru, dan mengoptimalkan kinerja. Ini seringkali menghasilkan SO yang lebih aman, stabil, dan fleksibel karena transparansi dan keterlibatan kolektif. Proyek-proyek seperti Android (berbasis Linux) dan berbagai distribusi Linux menunjukkan kekuatan model ini.



Gambar 1.1

D. Contoh Sistem Operasi Modern (Windows 11, macOS, Linux, Android, iOS)

Untuk memberikan gambaran yang lebih konkret, berikut adalah beberapa contoh sistem operasi modern yang saat ini mendominasi berbagai segmen pasar komputasi, masing-masing dengan karakteristik unik dan area fokusnya:

1. Windows 11:
    a. Platform: Desktop, Laptop, Tablet (khususnya untuk produktivitas).
    b. Karakteristik: Penerus Windows 10, menawarkan antarmuka pengguna yang dirancang ulang dengan fokus pada kesederhanaan dan produktivitas. Fitur-fitur modernnya mencakup Snap Layouts dan Snap Groups untuk *multi-tasking* yang lebih baik, integrasi Microsoft Teams yang mendalam,

dan yang paling revolusioner adalah kemampuan untuk menjalankan aplikasi Android secara *native* melalui Amazon Appstore. Keamanan ditingkatkan dengan persyaratan *Trusted Platform Module* (TPM) 2.0 dan *Secure Boot*.

c. Penggunaan: Sangat populer untuk penggunaan pribadi, *gaming*, produktivitas kantor, dan pengembangan *software*.

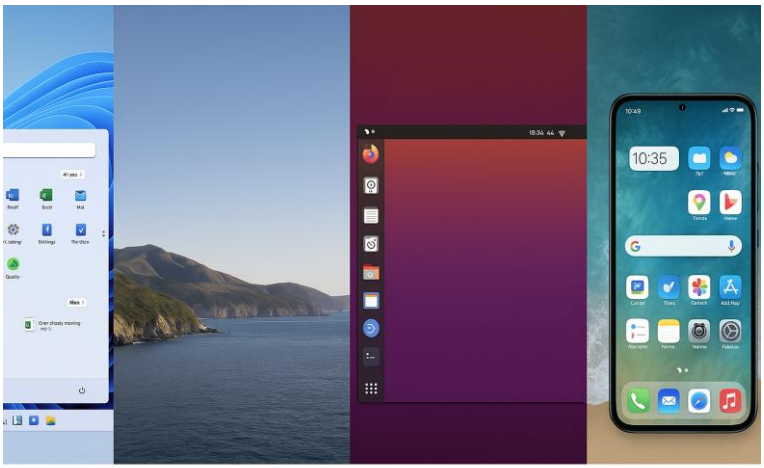
2. macOS:
   a. Platform: Desktop, Laptop (Produk Apple).
   b. Karakteristik: Dikenal dengan desain antarmuka yang elegan, stabilitas yang sangat baik, dan integrasi yang erat dalam ekosistem perangkat keras dan lunak Apple. macOS unggul dalam manajemen daya, grafis canggih, dan fitur keamanan serta privasi yang ketat. Pembaruan berkelanjutan membawa inovasi seperti fitur keamanan Gatekeeper dan *sandboxing* aplikasi yang ketat.
   c. Penggunaan: Populer di kalangan profesional kreatif, pengembang *software*, dan pengguna yang mencari pengalaman komputasi premium dan terintegrasi.
3. Linux:
   a. Platform: Server, Superkomputer, Desktop, Laptop, *Embedded Devices*, IoT.
   b. Karakteristik: Merupakan keluarga sistem operasi *open source* yang dibangun di atas kernel Linux. Kekuatan utamanya adalah fleksibilitas, skalabilitas, dan stabilitas luar biasa. Tersedia dalam berbagai "distribusi" (misalnya, Ubuntu, Fedora, Debian, Red Hat Enterprise Linux) yang disesuaikan untuk berbagai kebutuhan. Linux menjadi tulang punggung

internet, *cloud computing*, dan banyak infrastruktur *enterprise* karena keamanannya yang kuat dan efisiensi sumber dayanya.

c. Penggunaan: Dominan di lingkungan server dan *cloud*, banyak digunakan oleh pengembang, peneliti, dan pengguna yang mengutamakan kustomisasi dan kontrol.

4. Android:
   a. Platform: *Smartphone*, Tablet, Smart TV, Smartwatch, Mobil (Android Auto), Perangkat IoT.
   b. Karakteristik: Sistem operasi *mobile* paling dominan di dunia, dikembangkan oleh Google dan berbasis kernel Linux. Dikenal karena sifatnya yang *open source* (meskipun dengan lapisan *proprietary* Google Mobile Services), ekosistem aplikasi yang sangat luas (Google Play Store), dan fleksibilitas untuk disesuaikan oleh berbagai produsen perangkat. Android terus berinovasi dalam manajemen daya (Doze Mode), keamanan aplikasi (sandboxing, izin granular), dan fitur-fitur AI.
   c. Penggunaan: Utama untuk *smartphone* dan tablet, menjadi dasar bagi sebagian besar perangkat *mobile* non-Apple.
5. iOS:
   a. Platform: iPhone, iPad, iPod Touch.
   b. Karakteristik: Sistem operasi *mobile* eksklusif Apple, dikenal dengan antarmuka pengguna yang sangat intuitif dan mudah digunakan, performa yang mulus, dan fokus yang kuat pada privasi serta keamanan pengguna. iOS menerapkan kontrol ketat pada ekosistem aplikasinya melalui App Store, memastikan kualitas dan meminimalkan risiko *malware*. Integrasi mendalam dengan perangkat keras Apple dan layanan *cloud* (iCloud) memberikan pengalaman yang kohesif.

c. Penggunaan: Utama untuk *smartphone* dan tablet premium dari Apple.



Gambar 1.2

BAB 2: ARSITEKTUR DAN DESAIN SISTEM OPERASI MODERN

Desain arsitektur merupakan inti dari fungsionalitas dan kinerja sebuah sistem operasi. Sistem operasi modern, berbeda dengan pendahulunya, dirancang untuk menghadapi kompleksitas komputasi saat ini, termasuk kebutuhan akan skalabilitas, keamanan, efisiensi, dan kemampuan beradaptasi dengan lingkungan komputasi yang beragam. Bab ini akan mengulas prinsip-prinsip arsitektur dan desain fundamental yang membentuk sistem operasi modern, termasuk konsep modularitas dan mikrokernel, desain berbasis layanan, pendekatan kontainer dan virtualisasi, serta karakteristik khusus dari sistem operasi *real-time* dan *embedded*. Pemahaman mendalam tentang arsitektur ini krusial untuk mengapresiasi kapabilitas dan inovasi yang ada pada SO masa kini.

A. Modularitas dan Microkernel

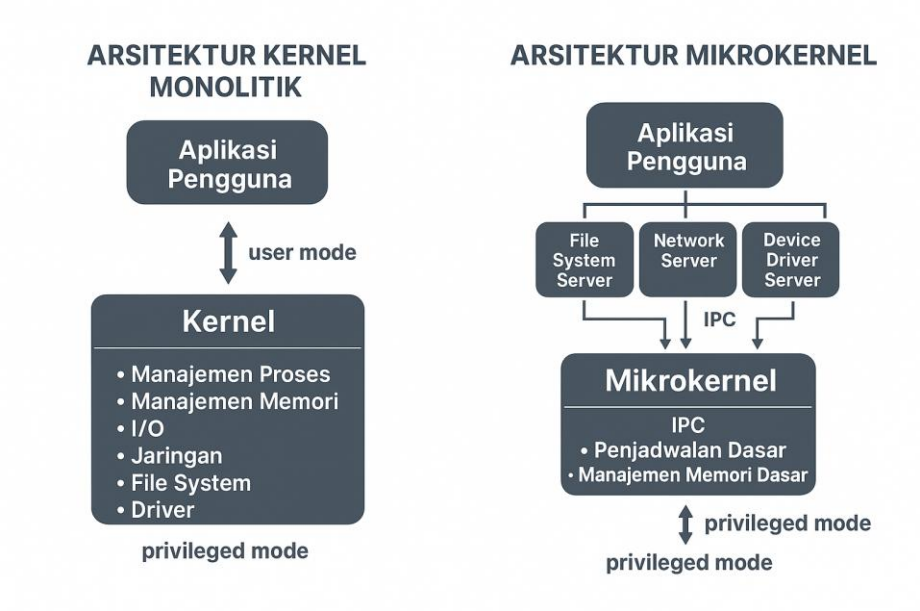
Arsitektur sebuah sistem operasi sangat menentukan stabilitas, keamanan, dan kemudahan pengembangannya. Secara historis, banyak sistem operasi awal mengadopsi desain monolitik, di mana seluruh layanan sistem inti—seperti manajemen proses, manajemen memori, sistem *file*, *device driver*, dan layanan jaringan—dikompilasi menjadi satu blok kode besar yang berjalan dalam ruang kernel (*kernel space*), yaitu bagian yang paling *privileged* dari sistem. Keuntungan utama dari arsitektur monolitik adalah kinerja yang tinggi karena semua komponen dapat berkomunikasi langsung tanpa *overhead* komunikasi antarproses. Namun, kelemahannya signifikan: kegagalan di satu komponen (misalnya, *bug* pada *device driver*) dapat

meruntuhkan seluruh kernel (*kernel panic*), menyebabkan sistem *crash*. Selain itu, sulit untuk melakukan pembaruan atau modifikasi pada satu komponen tanpa harus mengkompilasi ulang seluruh kernel.

1. Sebagai respons terhadap keterbatasan ini, arsitektur modularitas dan mikrokernel muncul sebagai paradigma desain kunci dalam sistem operasi modern.
2. Modularitas: Konsep modularitas menekankan pemisahan fungsionalitas sistem operasi ke dalam unit-unit yang lebih kecil dan terisolasi, yang disebut modul. Modul-modul ini dapat dimuat (*load*) atau dibongkar (*unload*) dari kernel saat sistem berjalan tanpa memerlukan *reboot* total. Kernel Linux, meskipun secara teknis sering disebut monolitik, sebenarnya sangat modular, memungkinkan penambahan atau penghapusan *device driver* atau fitur tertentu sebagai modul kernel yang dapat dimuat secara dinamis (*loadable kernel modules*). Ini meningkatkan fleksibilitas, mempermudah pemeliharaan, dan mengurangi ukuran kernel dasar.
3. Mikrokernel: Arsitektur mikrokernel adalah bentuk modularitas yang paling ekstrem. Dalam desain mikrokernel, kernel inti dijaga sekecil mungkin, hanya mencakup fungsi-fungsi paling esensial seperti:
4. Manajemen Komunikasi Antarproses (IPC - *Inter-Process Communication*): Mekanisme untuk proses-proses yang berbeda (termasuk server layanan) dapat saling berkomunikasi.
5. Manajemen Memori Tingkat Rendah: Pengelolaan ruang alamat virtual dasar dan perlindungan memori.
6. Penjadwalan Proses Dasar: Penjadwalan *thread* dan proses yang sangat primitif.

7. Semua layanan sistem operasi lainnya—seperti sistem *file*, *device driver*, *networking stack*, dan manajemen *hardware* yang lebih kompleks—dipindahkan dari ruang kernel ke ruang pengguna (*user space*) dan dijalankan sebagai proses-proses terpisah yang disebut server. Server-server ini berkomunikasi dengan mikrokernel dan satu sama lain melalui mekanisme IPC.
8. Kelebihan Mikrokernel:
    a. Stabilitas dan Keandalan yang Lebih Tinggi: Jika satu server layanan mengalami *crash* (misalnya, *device driver* kartu jaringan), hanya server tersebut yang terpengaruh, bukan seluruh kernel. Sistem operasi dapat memulihkan atau me-*restart* server yang *crash* tanpa *reboot* sistem.
    b. Keamanan yang Lebih Baik: Karena layanan berada di ruang pengguna, mereka memiliki *privilege* yang lebih rendah dan terisolasi satu sama lain. Ini membatasi kerusakan yang dapat ditimbulkan oleh *bug* atau serangan *malware* pada satu komponen.
    c. Fleksibilitas dan Kemudahan Pengembangan: Pengembang dapat menambah, menghapus, atau memodifikasi layanan sistem tanpa harus memodifikasi atau mengkompilasi ulang kernel inti. Ini memungkinkan pengembangan yang lebih cepat dan adaptasi yang lebih mudah terhadap perangkat keras atau fitur baru.
    d. Portabilitas: Kernel yang lebih kecil dan lebih terdefinisi dengan baik cenderung lebih mudah untuk di-*porting* ke arsitektur *hardware* yang berbeda.

9. Kekurangan Mikrokernel:
    a. Performa yang Berpotensi Lebih Rendah: Komunikasi antarproses melalui IPC memerlukan *context switch* dan *message passing*, yang menimbulkan *overhead* performa dibandingkan komunikasi langsung di arsitektur monolitik. Namun, inovasi dalam desain mikrokernel dan *hardware* modern telah banyak mengurangi *overhead* ini.
    b. Contoh sistem operasi yang menggunakan atau terinspirasi mikrokernel antara lain QNX (digunakan di otomotif dan industri), MINIX, dan sebagian dari arsitektur macOS (XNU kernel adalah *hybrid* yang menggabungkan elemen mikrokernel Mach dan kernel BSD monolitik).



Gambar 1.3

B. Desain Berbasis Layanan (Service-Oriented OS)

Melanjutkan konsep modularitas dan isolasi yang diperkenalkan oleh arsitektur mikrokernel, Desain Berbasis Layanan (Service-Oriented OS - SOOS) memperluas gagasan bahwa fungsionalitas sistem operasi harus disediakan sebagai kumpulan layanan yang terdefinisi dengan baik. Dalam SOOS, komponen-komponen sistem operasi (misalnya, manajemen *file*, manajemen *device*, keamanan, *networking*) tidak lagi dilihat sebagai bagian integral dari kernel monolitik yang tak terpisahkan, melainkan sebagai layanan independen yang dapat diakses melalui antarmuka standar.

Konsep ini mirip dengan arsitektur *microservices* dalam pengembangan aplikasi, di mana aplikasi besar dipecah menjadi layanan-layanan kecil yang independen dan berkomunikasi melalui API. Dalam konteks sistem operasi:

1. Abstraksi Fungsionalitas: Setiap layanan SO menawarkan fungsionalitas spesifik melalui API yang terdefinisi, menyembunyikan detail implementasi internal.
2. Isolasi Proses: Setiap layanan seringkali berjalan sebagai proses atau *thread* terpisah, terisolasi dari layanan lain dan dari kernel inti. Ini meningkatkan keamanan dan toleransi kesalahan. Jika satu layanan *crash*, layanan lain tetap berfungsi dan sistem secara keseluruhan tidak terganggu.
3. Komunikasi Melalui Pesan: Layanan-layanan ini berkomunikasi satu sama lain dan dengan aplikasi melalui mekanisme *message passing* atau panggilan prosedur jarak jauh (*Remote Procedure Call* - *RPC*), mirip dengan IPC pada mikrokernel.
4. Skalabilitas Dinamis: Karena layanan-layanan bersifat independen, beberapa instansi dari layanan yang sama dapat dijalankan secara paralel untuk menangani beban kerja yang

meningkat. Ini sangat relevan dalam lingkungan *cloud* atau terdistribusi.

5. Kemudahan Pemeliharaan dan Pembaruan: Layanan dapat diperbarui, diganti, atau ditambahkan tanpa memengaruhi layanan lain atau memerlukan *reboot* sistem. Ini mempercepat siklus pengembangan dan *deployment*.
6. Manfaat Desain Berbasis Layanan:
7. Peningkatan Keandalan: Isolasi layanan meminimalkan dampak kegagalan.
8. Keamanan yang Ditingkatkan: Permukaan serangan diperkecil karena layanan memiliki *privilege* yang minimal dan hanya berinteraksi melalui API yang terkontrol.
9. Fleksibilitas dan Kustomisasi: Memungkinkan pengembang untuk memilih dan mencampur layanan yang berbeda, atau bahkan mengembangkan layanan kustom, untuk memenuhi kebutuhan spesifik.
10. Dukungan untuk Komputasi Terdistribusi: Desain berbasis layanan secara inheren mendukung lingkungan terdistribusi, di mana layanan dapat berjalan pada node yang berbeda dalam jaringan.

Meskipun sedikit SO yang sepenuhnya *service-oriented* dari awal, banyak SO modern mengadopsi prinsip-prinsip desain ini dalam komponen-komponennya. Misalnya, dalam sistem operasi *cloud*, berbagai fungsi pengelolaan *virtual machine*, jaringan virtual, dan penyimpanan seringkali disediakan sebagai layanan terpisah.

C. Pendekatan Container dan Virtualisasi

Salah satu inovasi paling transformatif dalam arsitektur sistem operasi modern adalah kemampuan untuk menjalankan lingkungan komputasi terisolasi di atas *host* SO. Ini diwujudkan melalui virtualisasi dan kontainerisasi, yang keduanya bertujuan untuk menyediakan isolasi dan efisiensi, tetapi dengan pendekatan yang berbeda.

1. Konsep Virtualisasi: Virtualisasi memungkinkan satu perangkat keras fisik (*host machine*) untuk menjalankan beberapa sistem operasi virtual (*guest operating systems*), masing-masing berfungsi seperti mesin fisik yang mandiri. Ini dicapai dengan menggunakan lapisan perangkat lunak yang disebut hypervisor (juga dikenal sebagai *Virtual Machine Monitor - VMM*). Hypervisor bertanggung jawab untuk mengelola sumber daya *hardware* dan mendistribusikannya ke masing-masing *virtual machine* (VM).
2. Ada dua tipe utama hypervisor:
    a. Hypervisor Tipe 1 (Bare-Metal Hypervisor): Langsung berjalan di atas *hardware* fisik, tanpa memerlukan sistem operasi *host* di bawahnya. Hypervisor ini memiliki kendali langsung atas *hardware*, yang menghasilkan kinerja yang sangat tinggi dan keamanan yang kuat. Contoh: VMware ESXi, Microsoft Hyper-V, Citrix XenServer, KVM (pada Linux).
    b. Hypervisor Tipe 2 (Hosted Hypervisor): Berjalan sebagai aplikasi di atas sistem operasi *host* yang sudah ada. Meskipun lebih mudah dipasang dan digunakan untuk tujuan pengembangan atau pengujian, kinerjanya sedikit lebih rendah

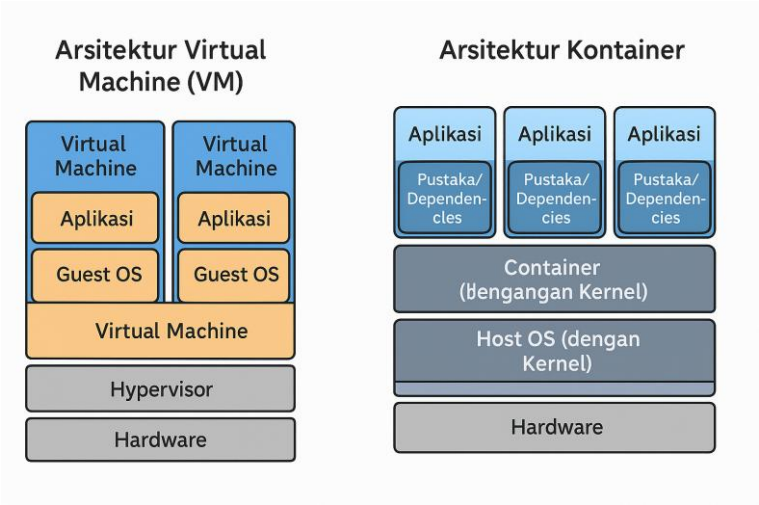
karena harus melalui lapisan SO *host*. Contoh: VirtualBox, VMware Workstation, VMware Fusion.

3. Setiap VM mencakup kernel sistem operasi *guest* yang lengkap, pustaka sistem, dan aplikasi. Ini membuatnya sangat terisolasi, tetapi juga memerlukan sumber daya yang signifikan untuk setiap VM.
4. Konsep Kontainerisasi: Kontainer menawarkan bentuk virtualisasi yang lebih ringan dan efisien. Berbeda dengan VM yang memvirtualisasikan seluruh mesin (*hardware* dan OS), kontainer memvirtualisasikan sistem operasi di tingkat proses. Artinya, semua kontainer yang berjalan di satu *host* SO berbagi kernel *host* yang sama, tetapi setiap kontainer memiliki ruang pengguna terisolasi sendiri, termasuk sistem *file* yang terpisah, pustaka, dan dependensi aplikasi.
5. Teknologi kontainer seperti Docker dan orkestrasi seperti Kubernetes memanfaatkan fitur kernel Linux (seperti *namespaces* untuk isolasi proses dan *cgroups* untuk alokasi sumber daya) untuk mencapai isolasi ini.

Perbandingan Virtual Machine vs. Container:

Fitur Kunci	Virtual Machine (VM)	Container
Tingkat Isolasi	Tinggi (Virtualisasi seluruh *hardware*)	Sedang (Virtualisasi OS di tingkat proses)
Sumber Daya	Membutuhkan lebih banyak sumber daya (CPU, RAM, Disk)	Lebih ringan, berbagi kernel *host*
Ukuran Image	Besar (termasuk OS *guest* lengkap)	Kecil (hanya aplikasi dan dependensinya)

Waktu Booting	Lebih lama (mem-boot OS *guest*)	Sangat cepat (hanya memulai proses aplikasi)
Portabilitas	Sangat portabel antar hypervisor	Sangat portabel antar *host* dengan kernel yang kompatibel
Ketergantungan	Masing-masing VM memiliki kernel OS sendiri	Berbagi kernel OS *host*
Kasus Penggunaan	Menjalankan OS yang berbeda, isolasi ketat, pengujian	Pengembangan, *deployment microservices*, CI/CD, skalabilitas cepat



Gambar 1.4

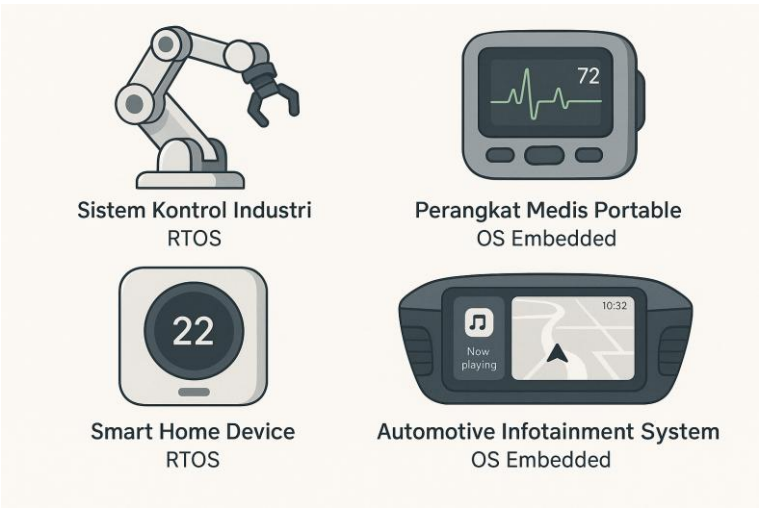
D. Sistem Operasi Real-Time dan Embedded

Selain sistem operasi umum untuk *desktop* dan *server*, ada kategori khusus yang dirancang untuk kebutuhan sangat spesifik: Sistem Operasi *Real-Time* dan Sistem Operasi *Embedded*.

1. Sistem Operasi Real-Time (RTOS): RTOS adalah jenis sistem operasi yang dirancang untuk menjamin respons terhadap peristiwa eksternal dalam batasan waktu yang ketat dan dapat diprediksi. Berbeda dengan SO umum yang mengutamakan *throughput* atau keadilan pembagian waktu CPU, RTOS memprioritaskan *timeliness* dan *determinism*.
    a. Hard Real-Time System: Memiliki batasan waktu yang sangat ketat dan mutlak. Kegagalan memenuhi tenggat waktu dapat menyebabkan kegagalan sistem yang katastropal (misalnya, sistem kontrol penerbangan, perangkat medis pendukung kehidupan).
    b. Soft Real-Time System: Memiliki tenggat waktu yang penting tetapi kegagalan sesekali untuk memenuhinya tidak menyebabkan bencana total, hanya penurunan kualitas layanan (misalnya, sistem *streaming* video, sistem kontrol *game*).
2. Karakteristik RTOS:
    a. Penjadwalan Preemtif Prioritas Tinggi: Memberikan prioritas tertinggi pada tugas *real-time* dan memungkinkannya untuk menginterupsi tugas berprioritas lebih rendah kapan saja.
    b. Latensi Rendah: Meminimalkan waktu yang dibutuhkan untuk merespons interupsi dan beralih antar tugas.
    c. Determinisme: Menjamin bahwa operasi akan diselesaikan dalam jangka waktu yang dapat diprediksi, bahkan di bawah beban tinggi.

d. Ukuran Minimal: Seringkali dirancang untuk memiliki *footprint* memori dan *disk* yang kecil.

3. Contoh RTOS meliputi QNX, VxWorks, FreeRTOS, dan RT-Linux.
4. Sistem Operasi Embedded: SO *embedded* adalah sistem operasi yang dirancang khusus untuk perangkat *embedded*—sistem komputasi yang tertanam di dalam perangkat yang lebih besar dan dirancang untuk satu atau beberapa fungsi spesifik. Perangkat ini biasanya memiliki sumber daya komputasi yang terbatas (memori, CPU, daya) dan seringkali tidak memiliki antarmuka pengguna grafis tradisional.
5. Karakteristik SO Embedded:
   a. Ukuran dan Footprint yang Sangat Kecil: Dioptimalkan untuk memori dan penyimpanan terbatas.
   b. Efisiensi Daya: Dirancang untuk beroperasi dengan konsumsi daya minimal.
   c. Kustomisasi Tinggi: Seringkali sangat disesuaikan dengan *hardware* spesifik perangkat.
   d. Keandalan dan Stabilitas: Penting untuk operasi jangka panjang tanpa *reboot*.
   e. Keamanan: Fitur keamanan yang dioptimalkan untuk perangkat keras terbatas dan lingkungan IoT.
6. SO *embedded* dapat berupa RTOS (jika membutuhkan *timeliness*) atau SO yang lebih umum yang dimodifikasi untuk lingkungan *embedded* (misalnya, versi khusus Linux untuk IoT seperti OpenWrt, atau Android *embedded*). Contoh perangkat yang menggunakan SO *embedded* termasuk *router* nirkabel, peralatan rumah tangga pintar, sistem otomotif, perangkat medis, dan sistem kontrol industri.


72
Sistem Kontrol Industri
RTOS
Perangkat Medis Portable
OS Embedded
22
Smart Home Device
RTOS
10:32
Now playing
Automotive Infotainment System
OS Embedded

Gambar 1.5

BAB 3: VIRTUALISASI DAN CLOUD COMPUTING

Dalam era komputasi modern, virtualisasi dan *cloud computing* bukan lagi sekadar jargon teknis, melainkan fondasi esensial yang telah merevolusi cara sistem operasi (SO) berinteraksi dengan perangkat keras dan bagaimana layanan komputasi disediakan. Kedua paradigma ini memungkinkan efisiensi sumber daya yang belum pernah ada sebelumnya, skalabilitas yang fleksibel untuk berbagai kebutuhan, serta ketahanan sistem yang jauh lebih baik dibandingkan model komputasi tradisional. Bab ini akan memandu pembaca untuk memahami secara mendalam konsep dasar virtualisasi, menjelaskan peran krusial *hypervisor* dalam memungkinkan fenomena ini, serta secara komprehensif membandingkan dua pendekatan virtualisasi yang paling dominan: *virtual machine* (VM) dan *container*. Selanjutnya, kita akan menyelami bagaimana sistem operasi modern beradaptasi dan berfungsi secara optimal dalam lingkungan virtual, serta bagaimana mereka terintegrasi erat dengan platform *cloud computing*. Bab ini akan diakhiri dengan eksplorasi model inovatif Sistem Operasi sebagai Layanan (OSaaS) yang semakin populer.

A. Konsep Virtualisasi

Virtualisasi adalah sebuah konsep transformatif dalam ilmu komputer yang melibatkan penciptaan representasi virtual, bukan fisik, dari sebuah sumber daya komputasi. Bayangkan sebuah komputer fisik tunggal yang dapat berperilaku seperti beberapa komputer yang terpisah secara independen. Itulah esensi virtualisasi. Ini memungkinkan berbagai sistem operasi, aplikasi, dan konfigurasi

lingkungan untuk berjalan secara bersamaan di atas satu set perangkat keras fisik yang sama, memaksimalkan utilisasi sumber daya dan mengurangi biaya infrastruktur.

1. Tujuan utama virtualisasi meliputi:
2. Konsolidasi Sumber Daya: Mengurangi jumlah *server* fisik yang dibutuhkan, yang berarti penghematan biaya *hardware*, energi (listrik dan pendinginan), serta ruang *data center*.
3. Isolasi: Setiap lingkungan virtual (VM atau *container*) terisolasi dari yang lain, sehingga masalah atau *crash* pada satu lingkungan tidak akan mempengaruhi yang lain.
4. Fleksibilitas: Memungkinkan *deployment* dan manajemen lingkungan komputasi yang lebih cepat dan mudah.
5. Portabilitas: Lingkungan virtual dapat dengan mudah dipindahkan antar *server* fisik yang berbeda.
6. Hypervisor (Type 1 & Type 2): Sang Arsitek Virtualisasi
7. Inti dari teknologi virtualisasi adalah hypervisor, sering juga disebut *Virtual Machine Monitor* (VMM). Hypervisor adalah lapisan perangkat lunak, *firmware*, atau *hardware* yang bertanggung jawab untuk menciptakan dan menjalankan *virtual machine*. Ini adalah "otak" di balik kemampuan untuk membagi sumber daya fisik dan mendistribusikannya secara aman ke VM yang berbeda.
8. Hypervisor Tipe 1 (Bare-Metal Hypervisor):
    a. Cara Kerja: Hypervisor tipe ini diinstal langsung pada perangkat keras fisik komputer, tanpa memerlukan sistem operasi *host* terlebih dahulu. Ini berarti hypervisor memiliki kontrol langsung dan penuh atas sumber daya *hardware*. Analogi sederhananya adalah, jika perangkat keras adalah tanah kosong, maka hypervisor tipe 1 adalah kontraktor utama

yang langsung membangun fondasi dan dinding-dinding rumah virtual (VM) di atasnya.

b. Keunggulan: Karena berinteraksi langsung dengan *hardware*, hypervisor Tipe 1 menawarkan kinerja yang sangat tinggi (mendekati kinerja *native*), stabilitas yang superior, dan keamanan yang lebih robust karena tidak ada lapisan SO *host* yang dapat menjadi titik kerentanan tambahan. Mereka sangat efisien dalam alokasi sumber daya.

c. Penerapan: Ini adalah pilihan dominan di lingkungan *data center* skala besar, *cloud computing* publik, dan infrastruktur *server enterprise* di mana kinerja dan keandalan adalah prioritas utama.

d. Contoh: VMware ESXi, Microsoft Hyper-V, Citrix XenServer, dan KVM (*Kernel-based Virtual Machine*, yang merupakan bagian dari kernel Linux dan mengubah Linux menjadi hypervisor Tipe 1).

9. Hypervisor Tipe 2 (Hosted Hypervisor):

a. Cara Kerja: Berbeda dengan Tipe 1, hypervisor Tipe 2 berjalan sebagai aplikasi perangkat lunak biasa di atas sistem operasi *host* yang sudah ada (misalnya, Windows, macOS, atau Linux). SO *host* inilah yang bertanggung jawab untuk mengelola *hardware* fisik, dan hypervisor Tipe 2 kemudian mengalokasikan sumber daya virtual dari *host* ke mesin virtual. Menggunakan analogi sebelumnya, jika *hardware* adalah tanah kosong dan SO *host* adalah rumah yang sudah dibangun, maka hypervisor tipe 2 adalah sebuah ruangan khusus di dalam rumah tersebut yang kemudian dibagi lagi menjadi "ruang-ruang virtual" (VM).

b. Keunggulan: Lebih mudah dipasang dan digunakan untuk tujuan pengembangan, pengujian *software*, atau penggunaan pribadi di mana pengguna ingin menjalankan beberapa SO di PC mereka tanpa mengubah konfigurasi *boot* utama. Fleksibel karena dapat memanfaatkan fitur-fitur SO *host*.
c. Kekurangan: Kinerja cenderung sedikit lebih rendah dibandingkan Tipe 1 karena adanya lapisan SO *host* yang menjadi perantara antara hypervisor dan *hardware* fisik. Keamanan juga sedikit lebih rentan karena ketergantungan pada keamanan SO *host*.
d. Contoh: VirtualBox, VMware Workstation, VMware Fusion.

10. Virtual Machine (VM) vs. Container: Dua Pendekatan Isolasi yang Berbeda
11. Meskipun VM dan *container* sama-sama bertujuan untuk menyediakan lingkungan terisolasi untuk aplikasi, mereka melakukannya dengan cara yang fundamental berbeda di tingkat arsitektur. Pemahaman perbedaan ini sangat penting dalam memilih teknologi yang tepat untuk kebutuhan spesifik.
12. Virtual Machine (VM): Virtualisasi Tingkat Hardware Penuh
    a. Definisi: VM adalah emulasi lengkap dari sebuah komputer fisik. Setiap VM bertindak seperti komputer mandiri dengan komponen *hardware* virtualnya sendiri (CPU virtual, RAM virtual, *disk* virtual, *network adapter* virtual).
    b. Struktur: Di dalam setiap VM, terdapat sistem operasi *guest* yang lengkap (termasuk kernel, pustaka sistem, dan dependensi lainnya) yang berjalan di atas *hardware* virtual yang disediakan oleh *hypervisor*. Di atas SO *guest* inilah aplikasi diinstal.

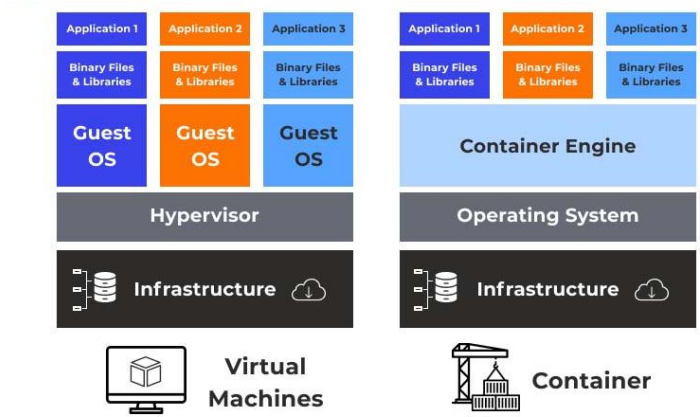
c. Isolasi: VM menawarkan tingkat isolasi yang sangat kuat. Karena setiap VM memiliki kernel SO-nya sendiri dan sumber daya virtual yang dialokasikan secara independen, masalah atau bahkan *malware* di satu VM akan sangat sulit untuk menyebar ke VM lain atau *host*. Ini seperti memiliki beberapa komputer fisik yang terpisah di dalam satu kotak.
d. Konsumsi Sumber Daya: VM cenderung membutuhkan sumber daya yang lebih besar (CPU, RAM, *disk space*) karena setiap VM harus memuat seluruh *instance* SO *guest* dari nol.
e. Waktu Booting: Waktu yang dibutuhkan untuk memulai VM lebih lama karena proses *booting* SO *guest* harus diselesaikan.
f. Portabilitas: VM sangat portabel antar *hypervisor* yang kompatibel, memungkinkan migrasi lingkungan aplikasi yang mulus.
g. Kasus Penggunaan Ideal: Menjalankan sistem operasi yang berbeda pada satu *hardware* fisik (misalnya, Windows di Linux), pengujian *software* di lingkungan yang terisolasi sepenuhnya, konsolidasi *server* yang membutuhkan isolasi maksimum.

13. Container: Virtualisasi Tingkat Sistem Operasi (Ringan)
    a. Definisi: Kontainer adalah bentuk virtualisasi yang lebih ringan dan efisien yang beroperasi pada tingkat sistem operasi. Ini memungkinkan banyak "lingkungan terisolasi" untuk berjalan di atas satu *host* SO yang sama.
    b. Struktur: Semua kontainer yang berjalan pada satu *host* berbagi kernel *host* SO yang sama. Setiap kontainer hanya berisi aplikasi dan dependensinya (pustaka, *runtime*, *configuration files*) yang diperlukan untuk menjalankan aplikasi tersebut, tanpa kernel SO *guest* yang terpisah.

c. Isolasi: Meskipun berbagi kernel *host*, kontainer menggunakan fitur-fitur kernel SO *host* seperti namespaces (untuk mengisolasi proses, *network interface*, *mount points*) dan cgroups (*control groups* untuk mengalokasikan dan membatasi sumber daya CPU, memori, I/O) untuk menciptakan lingkungan yang terisolasi. Isolasi ini kuat untuk sebagian besar kasus, tetapi secara teoritis sedikit kurang ketat dibandingkan VM karena berbagi kernel.
d. Konsumsi Sumber Daya: Kontainer jauh lebih ringan dan membutuhkan lebih sedikit sumber daya dibandingkan VM. Mereka tidak perlu mengalokasikan RAM atau CPU untuk kernel SO *guest* yang terpisah.
e. Waktu Booting: Kontainer sangat cepat untuk di-*boot* (hanya dalam hitungan detik) karena mereka tidak perlu melalui proses *booting* SO penuh, hanya memulai proses aplikasi.
f. Portabilitas: Kontainer sangat portabel antar *host* yang memiliki kernel SO yang kompatibel dan *container runtime* yang sama (misalnya, Docker).
g. Kasus Penggunaan Ideal: Pengembangan dan *deployment microservices*, *Continuous Integration/Continuous Deployment* (CI/CD), *serverless computing*, dan aplikasi yang membutuhkan skalabilitas cepat.

Fitur Kunci	Virtual Machine (VM)	Container
Lapisan Virtualisasi	Di atas *hardware* (oleh *hypervisor*)	Di atas SO *host* (oleh *container runtime*)
Isolasi	Tinggi (setiap VM memiliki kernel OS	Sedang (berbagi kernel OS *host*)

	sendiri)	
Konsumsi Sumber Daya	Membutuhkan lebih banyak (duplikasi OS kernel)	Lebih ringan (berbagi kernel OS *host*)
Ukuran Image	Besar (GBs, termasuk OS *guest* lengkap)	Kecil (MBs, hanya aplikasi dan dependensinya)
Waktu Booting	Lebih lama (mem-boot OS *guest* penuh)	Sangat cepat (hanya memulai proses aplikasi)
Portabilitas	Antar *hypervisor* yang kompatibel	Antar *host* dengan kernel OS yang kompatibel
Dependensi	Masing-masing VM memiliki kernel OS sendiri	Berbagi kernel OS *host*
Kasus Penggunaan	Menjalankan OS yang berbeda, isolasi ketat, *legacy apps*	*Microservices*, CI/CD, skalabilitas cepat, pengembangan agnostik OS



Gambar 1.6 *(Deskripsi Gambar: Dua diagram berdampingan. Diagram pertama berjudul "Arsitektur Virtual Machine (VM)". Tunjukkan lapisan "Hardware" di paling bawah. Di atas Hardware, gambarlah "Hypervisor" (misalnya VMware ESXi, Hyper-V). Di atas Hypervisor, gambarlah beberapa blok terpisah, masing-masing berlabel "Virtual Machine". Di dalam setiap blok Virtual Machine, tunjukkan "Guest OS (Kernel, Pustaka)" dan di atasnya "Aplikasi". Gunakan panah untuk menunjukkan bahwa Hypervisor mengelola Hardware.

Diagram kedua berjudul "Arsitektur Kontainer". Tunjukkan lapisan "Hardware" di paling bawah. Di atas Hardware, gambarlah "Host OS (dengan Kernel)" (misalnya Linux Kernel). Di atas Host OS, gambarlah lapisan "Container Runtime" (misalnya Docker Daemon). Di atas Container Runtime, gambarlah beberapa blok terpisah, masing-masing berlabel "Container". Di dalam setiap blok Container, tunjukkan hanya "Aplikasi" dan "Pustaka/Dependencies" mereka, tanpa Guest OS terpisah. Gunakan panah untuk menunjukkan bahwa Container Runtime berinteraksi dengan Host OS Kernel.

B. Sistem Operasi dalam Lingkungan Virtual

Ketika sebuah sistem operasi berjalan di dalam sebuah lingkungan virtual, ia menjadi "tamu" (*guest*) di atas infrastruktur yang dikelola oleh *hypervisor*. Adaptasi ini mengubah banyak aspek dari bagaimana SO berinteraksi dengan *hardware* dan bagaimana ia dikelola.

Optimasi untuk Lingkungan Virtual: SO modern dirancang untuk mengenali dan beradaptasi dengan lingkungan virtual. Vendor *hypervisor* menyediakan "alat integrasi" (misalnya, VMware Tools untuk VMware, Hyper-V Integration Services untuk Microsoft

Hyper-V) yang diinstal di dalam SO *guest*. Alat-alat ini berisi *driver* yang disebut driver paravirtualisasi (pv-driver). *Driver* ini memungkinkan SO *guest* untuk berkomunikasi lebih efisien dengan *hypervisor* daripada mencoba berinteraksi langsung dengan *hardware* fisik yang divirtualisasi. Ini mengurangi *overhead* dan meningkatkan kinerja I/O (disk, jaringan) serta manajemen CPU. Tanpa *driver* ini, SO *guest* mungkin harus menggunakan emulasi *hardware* yang lebih lambat.

1. Manajemen Sumber Daya Virtual:
    a. CPU: *Hypervisor* mengalokasikan siklus CPU dari prosesor fisik ke setiap VM. Meskipun SO *guest* "melihat" sejumlah inti CPU virtual, *hypervisor* adalah yang bertanggung jawab untuk menjadwalkan kapan instruksi dari VM tersebut benar-benar dieksekusi pada inti fisik. Teknik seperti *CPU overcommitment* memungkinkan penyedia *cloud* mengalokasikan lebih banyak inti CPU virtual daripada yang ada secara fisik, dengan asumsi bahwa tidak semua VM akan menggunakan 100% CPU secara bersamaan.
    b. Memori: *Hypervisor* mengelola alokasi memori fisik ke VM. Teknik seperti memory ballooning memungkinkan *hypervisor* untuk merebut kembali memori yang tidak digunakan dari VM yang sedang berjalan dan mengalokasikannya ke VM lain yang membutuhkan. SO *guest* akan "berpikir" memorinya berkurang dan akan melakukan *swapping* ke *disk* virtualnya, yang kemudian diatur oleh *hypervisor*. Ada juga teknik seperti *memory deduplication* (juga dikenal sebagai *page sharing*) di mana *hypervisor* dapat mengidentifikasi halaman memori yang identik di beberapa VM dan menyimpannya hanya sekali di RAM fisik, menghemat ruang.

c. I/O: *Hypervisor* mengintersep semua permintaan I/O dari SO *guest* dan menerjemahkannya ke *hardware* fisik. Dengan *driver* paravirtualisasi, komunikasi ini jauh lebih efisien dibandingkan emulasi *hardware* tradisional.

2. Manfaat Operasional:
   a. Snapshotting dan Cloning: Kemampuan untuk mengambil *snapshot* (titik pemulihan) dari VM dan membuat *clone* (salinan identik) dari VM sangat mempercepat pengembangan, pengujian, dan *deployment* lingkungan.
   b. Live Migration: SO *guest* dapat dipindahkan dari satu *server* fisik ke *server* fisik lain tanpa *downtime* yang signifikan, sebuah fitur krusial untuk pemeliharaan *server* tanpa mengganggu layanan.
   c. Fault Tolerance: Dalam beberapa konfigurasi *hypervisor*, VM dapat secara otomatis di-*restart* atau di-*failover* ke *host* lain jika *host* utamanya mengalami kegagalan, meningkatkan ketersediaan layanan.
   d. Resource Pooling: Sumber daya dari beberapa *server* fisik dapat digabungkan menjadi satu "kolam" besar, dan VM dapat dialokasikan dari kolam ini secara dinamis, meningkatkan efisiensi dan fleksibilitas manajemen.
3. Tantangan: Meskipun banyak manfaat, ada tantangan dalam menjalankan SO di lingkungan virtual, seperti *overhead* kinerja (meskipun minimal dengan paravirtualisasi), kompleksitas manajemen *hypervisor* sendiri, dan *licensing* SO yang perlu disesuaikan untuk lingkungan virtual.

C. Integrasi Sistem Operasi dengan Platform Cloud

Cloud computing pada dasarnya adalah perluasan dari virtualisasi, di mana sumber daya komputasi (server, penyimpanan, jaringan, SO) disediakan sebagai layanan melalui internet oleh penyedia pihak ketiga (misalnya, AWS, Azure, Google Cloud). Sistem operasi adalah jantung dari hampir setiap layanan *cloud*.

1. SO sebagai Komponen IaaS (Infrastructure as a Service): Di lapisan IaaS, pengguna mendapatkan akses ke infrastruktur komputasi virtual, yang paling sering berupa *virtual machine* (VM) yang dapat dikonfigurasi. Pengguna memiliki kontrol penuh atas sistem operasi yang diinstal di VM ini. Penyedia *cloud* menawarkan berbagai "gambar" atau "template" SO (*Amazon Machine Images/AMI* di AWS, *Virtual Machine Images* di Azure/GCP) yang siap digunakan (misalnya, berbagai distribusi Linux seperti Ubuntu, CentOS, Red Hat, atau versi Windows Server). Pengguna bertanggung jawab untuk menginstal aplikasi, mengkonfigurasi SO, dan melakukan *patching* serta *update* keamanan pada SO di dalam VM mereka. Ini sangat mirip dengan mengelola *server* fisik, tetapi dengan fleksibilitas dan skalabilitas *cloud*.
2. SO sebagai Komponen PaaS (Platform as a Service): Pada lapisan PaaS, penyedia *cloud* tidak hanya menyediakan infrastruktur dasar tetapi juga lingkungan *runtime* lengkap yang diperlukan untuk mengembangkan, menjalankan, dan mengelola aplikasi. Ini berarti sistem operasi yang mendasari dan *middleware* (misalnya, *web server*, *database*) sepenuhnya dikelola oleh penyedia *cloud*. Pengembang hanya perlu fokus pada kode aplikasi mereka dan menyebarkannya ke *platform* PaaS. Contohnya adalah Google App Engine, Azure App Service, atau Heroku. Pengguna tidak

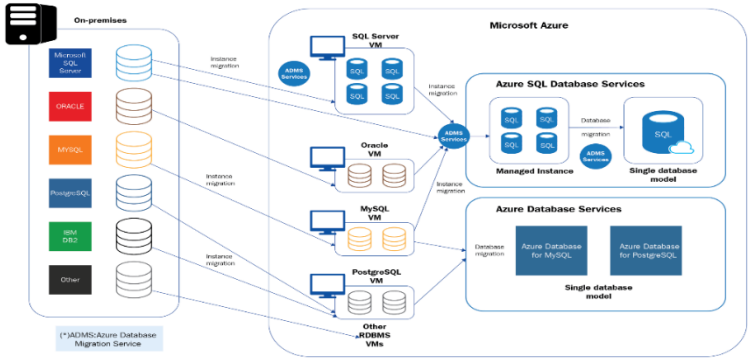
perlu khawatir tentang sistem operasi di balik layanan ini. Meskipun demikian, SO yang mendasarinya (seringkali Linux atau Windows Server yang dioptimalkan) adalah tulang punggung dari *platform* ini.

3. SO sebagai Komponen SaaS (Software as a Service): Di lapisan SaaS, penyedia *cloud* mengelola seluruh *stack* aplikasi, mulai dari *hardware*, sistem operasi, *middleware*, hingga aplikasi itu sendiri. Pengguna hanya mengakses aplikasi melalui *browser* web atau klien tipis (thin client). Contohnya adalah Microsoft 365, Google Workspace, atau Salesforce. Pengguna sama sekali tidak perlu berinteraksi atau mengelola sistem operasi yang menjalankan layanan-layanan ini. Dari perspektif pengguna akhir, SO yang mendasari tidak terlihat.

Otomatisasi dan Manajemen SO di Cloud: Integrasi SO dengan platform *cloud* sangat bergantung pada otomatisasi. Penyedia *cloud* menyediakan alat dan API untuk:

1. *Provisioning*: Membuat *instance* SO virtual dengan cepat.
2. *Scaling*: Otomatis menambah atau mengurangi jumlah *instance* SO berdasarkan beban kerja.
3. *Monitoring*: Memantau kinerja dan kesehatan SO.
4. *Patching dan Updating*: Otomatisasi proses pembaruan keamanan dan fitur SO.
5. *Configuration Management*: Mengelola konfigurasi SO secara konsisten di seluruh *instance* (misalnya, dengan alat seperti Ansible, Puppet, Chef). Konsep "Infrastruktur Immutabel" (Immutable Infrastructure) menjadi populer di *cloud*, di mana setelah sebuah *instance* VM atau *container* dengan SO di-*deploy*, ia tidak pernah dimodifikasi. Jika ada pembaruan atau perubahan, *instance* baru dengan versi SO yang telah diperbarui dibuat dan

instance lama dihentikan. Ini meningkatkan konsistensi dan mengurangi risiko "konfigurasi *drift*".



(Deskripsi Gambar: Diagram piramida atau tumpukan (stack) yang menggambarkan model layanan cloud IaaS, PaaS, dan SaaS dari bawah ke atas.

- Paling Bawah (IaaS): Lapisan "Physical Hardware", di atasnya "Virtualization/Hypervisor", dan di atasnya "Operating System (Dikelola Pengguna)". Di samping OS, tunjukkan "Aplikasi Pengguna".
- Lapisan Tengah (PaaS): Di atas IaaS, tunjukkan "Operating System", "Middleware", "Runtime" (semua ini Dikelola Penyedia). Di atasnya, tunjukkan "Aplikasi Pengguna".
- Paling Atas (SaaS): Di atas PaaS, tunjukkan "Aplikasi" (Dikelola Penyedia). Pengguna hanya berinteraksi dengan lapisan ini. Gunakan panah untuk menunjukkan kontrol: di IaaS, pengguna mengelola OS ke atas; di PaaS, pengguna mengelola aplikasi; di SaaS, pengguna hanya mengonsumsi aplikasi. Beri label "Dikelola Pengguna" dan "Dikelola Penyedia" pada setiap lapisan yang relevan untuk memperjelas tanggung jawab.)*

Cloud computing adalah model pengiriman layanan komputasi (termasuk *server*, penyimpanan, *database*, jaringan, *software*,

analitik, dan intelijen) melalui internet ("awan") dengan model bayar sesuai penggunaan. Sistem operasi adalah komponen fundamental dari setiap lapisan *cloud*.

1. SO sebagai Bagian dari IaaS (Infrastructure as a Service): Di lapisan IaaS, penyedia *cloud* menawarkan infrastruktur komputasi virtual, termasuk *virtual machine*. Pengguna memilih sistem operasi yang diinginkan (misalnya, berbagai distribusi Linux, Windows Server) dari daftar *image* yang tersedia (*Amazon Machine Images* di AWS, *Virtual Machine Images* di Azure/GCP) dan menyebarkannya ke dalam lingkungan *cloud* mereka. Dalam skenario ini, SO dipertimbangkan sebagai bagian dari "infrastruktur" yang dikelola oleh pengguna, memberikan kontrol penuh atas konfigurasi SO.
2. SO sebagai Bagian dari PaaS (Platform as a Service): Pada lapisan PaaS, penyedia *cloud* tidak hanya menyediakan infrastruktur tetapi juga lingkungan *runtime* untuk pengembangan dan *deployment* aplikasi. Ini berarti SO dan *middleware* yang mendasarinya sudah dikelola oleh penyedia *cloud*. Pengembang hanya perlu fokus pada kode aplikasi mereka. Meskipun pengguna tidak berinteraksi langsung dengan SO, SO yang mendasarinya (misalnya, Linux atau Windows Server) adalah fondasi bagi platform PaaS seperti Google App Engine, Azure App Service, atau AWS Elastic Beanstalk.
3. SO sebagai Bagian dari SaaS (Software as a Service): Di lapisan SaaS, penyedia *cloud* mengelola seluruh *stack* aplikasi, termasuk SO, infrastruktur, dan aplikasi itu sendiri. Pengguna hanya mengakses aplikasi melalui *browser* atau klien tipis (thin client). Contohnya adalah Microsoft 365, Google Workspace, atau

Salesforce. Pengguna tidak perlu khawatir tentang sistem operasi yang menjalankan aplikasi ini.

4. Manajemen SO di Cloud: Integrasi SO dengan platform *cloud* mencakup alat otomatisasi untuk *provisioning*, *scaling*, *monitoring*, dan *patching* SO virtual. *Cloud providers* menyediakan API dan layanan untuk mengelola siklus hidup SO, memastikan keamanan, dan ketersediaan tinggi. Konsep *immutable infrastructure*, di mana VM atau kontainer dengan SO tidak pernah dimodifikasi setelah *deployment* tetapi diganti dengan versi baru saat *update*, menjadi umum untuk meningkatkan konsistensi dan keandalan.

D. Sistem Operasi sebagai Layanan (OSaaS)

Sistem Operasi sebagai Layanan (OSaaS) adalah sebuah evolusi dari model *cloud computing* di mana akses ke lingkungan sistem operasi, beserta aplikasi dan layanan terkait, disediakan dan dikelola sepenuhnya sebagai layanan melalui internet. Berbeda dengan IaaS di mana pengguna masih bertanggung jawab atas pengelolaan SO, atau PaaS di mana SO tersembunyi di balik *platform*, OSaaS menawarkan pengalaman di mana *user* tidak perlu memikirkan instalasi, pembaruan, atau *maintenance* SO sama sekali.

1. Konsep dan Karakteristik: Dalam model OSaaS, pengguna berlangganan layanan yang memungkinkan mereka untuk mengakses dan menggunakan lingkungan desktop lengkap yang berbasis *cloud*. SO dan aplikasi berjalan di server *cloud*, dan pengguna berinteraksi melalui *thin client* (misalnya, *browser* web atau aplikasi desktop ringan) yang hanya menampilkan *stream* video dari desktop jarak jauh dan mengirimkan *input* (klik, ketikan) kembali ke server.

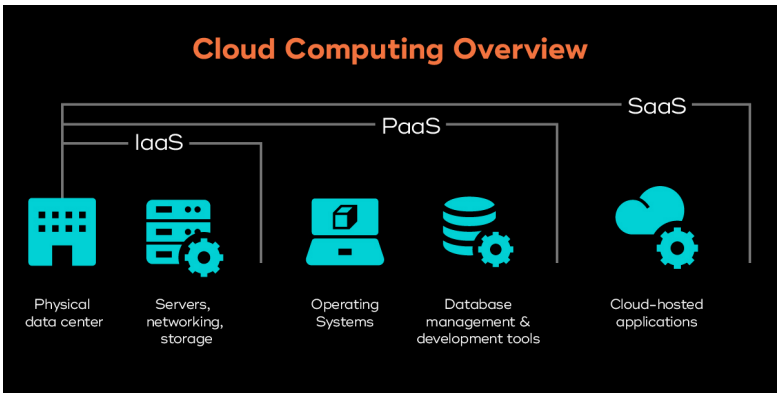
Karakteristik utama OSaaS meliputi:

a. Akses Anywhere, Anytime: Pengguna dapat mengakses lingkungan SO mereka dari perangkat apa pun (laptop, tablet, *smartphone*) dengan koneksi internet.
b. Manajemen oleh Penyedia: Instalasi, *patching*, *update*, keamanan, dan *backup* SO sepenuhnya ditangani oleh penyedia layanan. Ini mengurangi beban operasional bagi pengguna atau organisasi.
c. Skalabilitas On-Demand: Sumber daya (CPU, RAM) untuk lingkungan OSaaS dapat disekalakan secara dinamis oleh penyedia sesuai kebutuhan pengguna.
d. Efisiensi Biaya: Pengguna tidak perlu membeli lisensi SO atau *hardware* yang mahal, cukup membayar biaya berlangganan.

2. Contoh dan Penerapan: Salah satu contoh nyata OSaaS adalah Windows 365 dari Microsoft, yang menyediakan "PC Awan" (Cloud PC) yang dapat di-*stream* ke perangkat apa pun. Pengguna dapat memiliki Windows desktop yang dipersonalisasi dan aplikasi mereka tersedia secara instan dari *cloud*. Contoh lain termasuk layanan *desktop-as-a-service* (DaaS) dari penyedia *cloud* lainnya yang menawarkan lingkungan *desktop* virtual.
3. OSaaS sangat cocok untuk:
   a. Pekerja *remote* atau *hybrid* yang membutuhkan akses konsisten ke lingkungan kerja mereka dari berbagai lokasi dan perangkat.
   b. Institusi pendidikan yang ingin menyediakan lingkungan komputasi standar untuk siswa.
   c. Bisnis kecil dan menengah yang ingin mengurangi biaya TI dan kompleksitas manajemen infrastruktur.

d. Lingkungan dengan kebutuhan keamanan tinggi, di mana data tidak pernah meninggalkan *data center*.

(Deskripsi Gambar: Dua diagram berdampingan. Diagram pertama berjudul "Arsitektur Virtual Machine (VM)" menunjukkan lapisan "Hardware" di paling bawah, di atasnya ada "Hypervisor". Di atas Hypervisor ada beberapa blok "Virtual Machine", dan setiap blok VM berisi "Guest OS" dan "Aplikasi". Diagram kedua berjudul "Arsitektur Kontainer" menunjukkan lapisan "Hardware" di paling bawah, di atasnya ada "Host OS (dengan Kernel)". Di atas Host OS ada lapisan "Container Runtime" (misalnya Docker). Di atas Container Runtime ada beberapa blok "Container", dan setiap blok Container hanya berisi "Aplikasi" dan "Pustaka/Dependencies" mereka, tanpa OS guest terpisah, semua berbagi kernel host. Sertakan keterangan untuk setiap bagian diagram.)



(Deskripsi Gambar: Diagram piramida atau tumpukan (stack) yang menggambarkan model layanan cloud IaaS, PaaS, dan SaaS. Di bagian paling bawah (IaaS) tunjukkan "Hardware", di atasnya "Virtualisasi", dan di atasnya "OS (User-managed)". Di lapisan tengah (PaaS) tunjukkan "OS", "Middleware", "Runtime" (Provider-managed). Di lapisan paling atas (SaaS) tunjukkan "Aplikasi" (Provider-managed). Gunakan ikon atau label untuk menunjukkan di mana SO berinteraksi di setiap lapisan.)

(Deskripsi Gambar: Diagram yang menunjukkan seorang "Pengguna" yang berinteraksi dengan "Perangkat (Laptop/Tablet/Smartphone)" tipis. Dari perangkat tersebut, panah mengarah ke "Internet/Cloud". Di dalam cloud, gambarlah "Server" yang menjalankan "Sistem Operasi" dan "Aplikasi". Tunjukkan panah dua arah antara perangkat pengguna dan server di cloud, melambangkan streaming desktop dan input. Berikan keterangan bahwa OS dan aplikasi "Dikelola oleh Penyedia Layanan".)

BAB 4: SISTEM OPERASI MOBILE DAN PERANGKAT RINGAN

Bab ini akan membahas secara mendalam mengenai sistem operasi (OS) mobile dan karakteristik perangkat ringan yang menggunakannya. Kita akan menjelajahi ciri khas OS mobile, bagaimana mereka mengelola daya dan konektivitas, tantangan fragmentasi dan keamanan yang dihadapi, serta perbandingan antara dua OS mobile paling dominan: Android dan iOS.

A. Ciri Khas OS Mobile

Sistem operasi mobile dirancang dari awal dengan asumsi batasan dan fitur unik yang tidak ditemukan pada SO *desktop* tradisional. Ciri khas ini membentuk fundamental desain dan fungsionalitasnya:

- Antarmuka Pengguna Berbasis Sentuhan (Touch-Centric UI): Ini adalah perbedaan paling mencolok. OS mobile didesain untuk interaksi langsung melalui layar sentuh *multi-touch*, bukan *mouse* dan *keyboard*. Ini melibatkan elemen UI yang besar, *gesture* intuitif (cubit untuk *zoom*, *swipe* untuk navigasi), dan *keyboard* virtual.
- Optimalisasi Daya (Power Efficiency): Perangkat mobile sangat bergantung pada baterai. Oleh karena itu, manajemen daya adalah prioritas utama. OS mobile mengimplementasikan berbagai teknik untuk menghemat baterai, seperti:
    - Manajemen CPU Dinamis: Mengurangi frekuensi CPU atau mematikannya saat tidak diperlukan.

    - Manajemen Memori Agresif: Menutup atau menangguhkan aplikasi di latar belakang secara agresif untuk mengosongkan RAM dan menghemat daya.
    - Mode Tidur Dalam (*Deep Sleep Modes*): Memasuki mode daya rendah ekstrem saat perangkat tidak aktif.
- Konektivitas Nirkabel yang Meluas: OS mobile dirancang untuk selalu terhubung, mendukung berbagai teknologi nirkabel secara *native*:
    - Jaringan Seluler (2G/3G/4G/5G): Konektivitas data dan suara di mana saja.
    - Wi-Fi: Konektivitas kecepatan tinggi di area lokal.
    - Bluetooth: Untuk *pairing* dengan perangkat *wearable*, *headphone*, atau aksesoris lainnya.
    - NFC (Near Field Communication): Untuk pembayaran nirsentuh dan pertukaran data jarak dekat.
- Integrasi Sensor yang Mendalam: Perangkat mobile dilengkapi dengan beragam sensor yang sangat terintegrasi dengan OS:
    - Akselerometer dan Giroskop: Untuk mendeteksi orientasi perangkat dan gerakan (misalnya, rotasi layar, *gaming*).
    - GPS (Global Positioning System): Untuk layanan lokasi dan navigasi.
    - Sensor Cahaya Sekitar: Untuk menyesuaikan kecerahan layar secara otomatis.
    - Sensor Sidik Jari/Pengenalan Wajah: Untuk otentikasi biometrik.
    - Barometer, Kompas Magnetik, dll.: Untuk data lingkungan tambahan.
- Manajemen Aplikasi dan Ekosistem Aplikasi: OS mobile beroperasi dalam ekosistem aplikasi yang ketat dan terpusat (App

Store, Google Play Store). OS menyediakan kerangka kerja untuk instalasi, pembaruan, dan *sandboxing* aplikasi.
    - *Sandboxing*: Setiap aplikasi berjalan dalam lingkungan terisolasi untuk mencegahnya mengakses data atau fungsi aplikasi lain tanpa izin eksplisit pengguna, serta melindungi sistem inti.
    - Model Izin (Permission Model): Pengguna secara granular mengontrol izin yang diberikan kepada setiap aplikasi (misalnya, akses kamera, kontak, lokasi).
- Ukuran Footprint yang Kecil dan Efisien: Mengingat keterbatasan *hardware* perangkat mobile (RAM, penyimpanan), OS mobile dirancang untuk memiliki *footprint* yang sangat kecil dan efisien dalam penggunaan sumber daya.
- Fokus pada Pengalaman Pengguna (*User Experience*): Prioritas tinggi pada responsivitas, *smooth scrolling*, dan transisi antarmuka yang mulus untuk pengalaman pengguna yang menyenangkan.

1. Ikon baterai yang sedang diisi atau indikator daya rendah, menunjukkan pentingnya efisiensi energi.
2. Ikon konektivitas (Wi-Fi, 5G, Bluetooth) yang menyala.
3. Simbol sensor (misalnya, giroskop, lokasi GPS) di sekitar perangkat mobile.
4. Tampilan prompt izin aplikasi (misalnya, "Izinkan aplikasi X mengakses lokasi Anda?"). Semua elemen harus mengarah ke *smartphone* atau tablet sebagai pusatnya.)*

B. Manajemen Daya dan Konektivitas

Daya tahan baterai dan konektivitas yang andal adalah dua faktor penentu utama keberhasilan perangkat mobile. Sistem operasi mobile memiliki mekanisme canggih untuk mengelola kedua aspek ini secara optimal.

- Manajemen Daya Lanjut:
    - Doze Mode (Android) / Low Power Mode (iOS): Ini adalah fitur di mana OS secara cerdas menunda aktivitas aplikasi latar belakang yang tidak penting ketika perangkat tidak bergerak dan layar mati untuk jangka waktu tertentu. Notifikasi jaringan, sinkronisasi, dan tugas *background* lainnya digabungkan dan diproses secara periodik dalam *maintenance window* singkat, lalu perangkat kembali tidur. Ini secara drastis mengurangi konsumsi daya saat perangkat tidak digunakan.
    - App Standby (Android): Jika aplikasi tidak digunakan selama beberapa waktu, OS akan menempatkannya dalam status *standby*, membatasi aksesnya ke sumber daya jaringan dan CPU. Aplikasi hanya akan aktif kembali saat pengguna meluncurkannya.

- Penjadwalan Tugas Bertenaga-Sadar: OS mengoptimalkan penjadwalan proses dan layanan agar *hardware* (terutama CPU dan radio) dapat memasuki mode daya rendah sesering mungkin. Ini termasuk mengumpulkan tugas-tugas kecil menjadi satu *burst* untuk meminimalkan waktu bangun dari tidur.
- Optimasi Layar: Layar adalah salah satu komponen paling haus daya. OS mobile secara aktif mengelola kecerahan adaptif (berdasarkan sensor cahaya sekitar), *timeout* layar otomatis, dan mode Always-On Display yang hemat daya (terutama pada layar OLED).
- Manajemen Komponen Hardware: OS mengelola daya ke komponen *hardware* lain seperti GPU, sensor, dan radio nirkabel, mematikan atau menguranginya saat tidak aktif digunakan.

- Manajemen Konektivitas yang Adaptif: SO mobile harus memastikan konektivitas yang mulus dan efisien di berbagai jenis jaringan.
  - Peralihan Jaringan Otomatis: OS secara cerdas beralih antara Wi-Fi dan jaringan seluler (misalnya, dari 4G ke 5G) berdasarkan kekuatan sinyal, kecepatan, dan ketersediaan, seringkali tanpa intervensi pengguna.
  - Hotspot Seluler dan Tethering: Kemampuan untuk berbagi koneksi internet perangkat mobile dengan perangkat lain melalui Wi-Fi, Bluetooth, atau USB, yang sepenuhnya dikelola oleh OS.
  - Virtual Private Network (VPN) Support: OS menyediakan dukungan *native* atau API untuk VPN, memungkinkan

pengguna untuk membuat koneksi jaringan yang aman dan terenkripsi.

- Optimasi Penggunaan Data: OS seringkali menyediakan fitur untuk memantau penggunaan data seluler dan memungkinkan pengguna untuk mengatur batas data atau membatasi penggunaan data latar belakang untuk aplikasi tertentu.
- Bluetooth Low Energy (BLE): Dukungan untuk BLE memungkinkan perangkat untuk terhubung dengan aksesori seperti *wearable* dengan konsumsi daya yang sangat rendah, memperluas ekosistem perangkat.

C. Fragmentasi dan Keamanan

Dua tantangan terbesar yang dihadapi oleh sistem operasi mobile, terutama Android, adalah fragmentasi dan keamanan.

- Fragmentasi: Fragmentasi mengacu pada beragamnya versi OS, ukuran layar, resolusi, spesifikasi *hardware*, dan modifikasi *software* (kulit UI, aplikasi *bloatware*) yang ada dalam satu ekosistem.
  - Fragmentasi Android: Ini adalah masalah yang sangat menonjol di Android. Ada banyak produsen perangkat yang berbeda, dan masing-masing dapat memodifikasi Android Open Source Project (AOSP) untuk perangkat mereka. Akibatnya, pembaruan OS seringkali tertunda atau bahkan tidak pernah sampai ke perangkat lama, menciptakan ekosistem di mana banyak perangkat menjalankan versi Android yang berbeda dan terkadang sudah usang. Hal ini menyulitkan pengembang aplikasi untuk memastikan aplikasi mereka berfungsi dengan baik di semua perangkat dan versi,

serta menyulitkan pengguna untuk mendapatkan fitur dan *patch* keamanan terbaru.
    - Fragmentasi iOS: iOS memiliki tingkat fragmentasi yang sangat rendah. Karena Apple mengontrol *hardware* dan *software*, mereka dapat mendorong pembaruan OS ke hampir semua perangkat iOS secara bersamaan dan konsisten. Ini memastikan sebagian besar pengguna memiliki versi OS terbaru dengan fitur dan keamanan terkini.
- Dampak Fragmentasi:
    - Pengalaman Pengguna yang Tidak Konsisten: Fitur-fitur baru OS mungkin tidak tersedia di semua perangkat.
    - Pengembangan Aplikasi yang Lebih Sulit: Pengembang harus menguji dan mendukung banyak versi OS dan konfigurasi *hardware*.
    - Risiko Keamanan yang Meningkat: Perangkat yang menjalankan versi OS lama seringkali tidak menerima *patch* keamanan, sehingga lebih rentan terhadap eksploitasi.
- Keamanan pada OS Mobile: Keamanan adalah aspek krusial mengingat data sensitif yang disimpan dan diproses di perangkat mobile. OS mobile modern memiliki berbagai mekanisme keamanan:
    - *Sandboxing* Aplikasi: Seperti yang disebutkan, setiap aplikasi berjalan dalam lingkungan terisolasi, mencegah satu aplikasi untuk mengakses atau merusak data aplikasi lain atau sistem inti tanpa izin eksplisit.
    - Model Izin yang Granular: Pengguna harus secara eksplisit memberikan izin kepada aplikasi untuk mengakses *hardware* atau data sensitif (kamera, mikrofon, lokasi, kontak). OS mobile modern semakin memperketat kontrol izin ini.

- *Secure Boot*: Proses *booting* diverifikasi secara kriptografis dari *firmware* ke kernel hingga sistem *file* untuk memastikan tidak ada *malware* yang mengubah komponen sistem saat *booting*.
- Enkripsi Data Penuh (Full-Disk Encryption): Sebagian besar perangkat mobile modern mengenkripsi seluruh penyimpanan secara *default*, melindungi data bahkan jika perangkat dicuri.
- Pembaruan Keamanan Reguler: OS mobile secara rutin menerima *patch* keamanan untuk mengatasi kerentanan yang baru ditemukan. Namun, fragmentasi dapat menghambat penyebaran *patch* ini di seluruh ekosistem.
- Hardware-Based Security: Pemanfaatan fitur keamanan *hardware* seperti *Trusted Platform Module* (TPM) atau Secure Enclave (di perangkat Apple) untuk menyimpan kunci kriptografi atau data biometrik secara aman.
- Verifikasi Aplikasi: Baik Google Play Protect (Android) maupun proses peninjauan App Store (iOS) bertujuan untuk memindai aplikasi dari *malware* sebelum atau sesudah diunduh oleh pengguna.

D. Perbandingan Android dan iOS

Android dan iOS adalah dua raksasa yang mendominasi pasar sistem operasi *mobile* global. Meskipun keduanya menawarkan fungsionalitas inti yang serupa (seperti *multitasking*, akses internet, kamera, dll.), filosofi desain, model bisnis, dan ekosistem mereka sangat berbeda, yang pada akhirnya membentuk pengalaman pengguna yang berbeda pula. Memahami perbedaan fundamental ini penting untuk mengapresiasi kekuatan dan kelemahan masing-masing platform.

Fitur Kunci	Android	iOS
Pengembang Utama	Google (sebagian besar *open source*)	Apple Inc. (*proprietary*)
Model Bisnis	Berbasis lisensi *open source* (AOSP) dengan layanan Google Mobile Services (GMS) sebagai *proprietary*. Google mendapatkan pendapatan dari periklanan, layanan, dan penjualan aplikasi.	Terintegrasi vertikal: *hardware* (iPhone, iPad) dan *software* (iOS) terikat. Apple mendapatkan pendapatan dari penjualan *hardware* dan komisi App Store.
Filosofi Desain	Terbuka & Fleksibel: Menekankan kustomisasi, pilihan perangkat yang luas dari berbagai produsen, dan interoperabilitas. Memberikan lebih banyak kontrol kepada pengguna.	Tertutup & Terintegrasi: Menekankan kesederhanaan, konsistensi, performa optimal, dan keamanan melalui kontrol ketat pada ekosistem *hardware* dan *software*.
Ekosistem Perangkat	Sangat luas, dari puluhan bahkan ratusan produsen (Samsung, Xiaomi, Oppo, dll.) dengan rentang harga dan spesifikasi yang sangat beragam.	Terbatas pada perangkat yang diproduksi Apple saja (iPhone, iPad).

Kustomisasi UI	Tinggi. Pengguna dapat mengubah *launcher*, *widget*, *icon pack*, *keyboard*, dan tema secara ekstensif. Produsen juga dapat membuat *skin* UI kustom (misalnya, Samsung One UI, Xiaomi MIUI).	Rendah. Kustomisasi terbatas pada *wallpaper*, penataan *icon*, dan *widget* yang disediakan. UI inti sangat konsisten di semua perangkat dan versi.
Distribusi Aplikasi	Google Play Store: Lebih terbuka. Pengembang dapat mengunggah aplikasi relatif lebih mudah. Juga mendukung *sideloading* (instalasi aplikasi dari luar toko resmi).	Apple App Store: Sangat ketat. Semua aplikasi melalui proses peninjauan yang ketat dari Apple untuk kualitas, keamanan, dan privasi. *Sideloading* sangat dibatasi (biasanya hanya untuk pengembangan).
Fragmentasi OS	Tinggi. Banyak versi Android yang berbeda beredar karena produsen perangkat dan operator telekomunikasi sering menunda atau tidak merilis pembaruan. Ini menyulitkan *developer* dan menjadi celah keamanan.	Rendah. Apple memiliki kontrol penuh atas *hardware* dan *software*, memungkinkan pembaruan OS dirilis secara konsisten dan cepat ke sebagian besar perangkat yang didukung.
Pembaruan	Tergantung produsen	Dikontrol penuh oleh Apple.

OS	perangkat dan operator. Seringkali lambat atau tidak tersedia untuk perangkat lama. Google berupaya mengatasi ini dengan Project Treble dan Mainline.	Pembaruan biasanya dirilis secara serentak ke semua perangkat yang didukung, memastikan pengguna mendapatkan fitur dan *patch* keamanan terbaru.
Keamanan	Mengandalkan *sandboxing*, model izin, Google Play Protect, dan *Secure Boot*. Namun, fragmentasi dapat menimbulkan celah keamanan karena perangkat lama tidak menerima *patch*. Risiko *malware* sedikit lebih tinggi karena keterbukaan platform.	Sangat kuat. Mengandalkan *sandboxing*, model izin yang ketat, *Secure Enclave* (untuk biometrik/kriptografi), proses peninjauan App Store yang ketat. Risiko *malware* lebih rendah karena ekosistem yang terkontrol.
Integrasi Ekosistem	Terintegrasi dengan layanan Google (Gmail, Maps, Drive, Photos) dan *hardware* dari berbagai produsen. Interoperabilitas antar *brand* bisa bervariasi.	Integrasi vertikal yang sangat erat dengan *hardware* Apple (iPhone, iPad, Mac, Apple Watch, AirPods) dan layanan Apple (iCloud, iMessage, FaceTime). Menawarkan pengalaman yang sangat mulus antar perangkat Apple.
Asisten	Google Assistant (sangat	Siri (terintegrasi dengan

Suara	terintegrasi dengan layanan Google dan pengetahuan web).	ekosistem Apple, fokus pada tugas-tugas perangkat).
Ketersediaan	Berbagai segmen pasar, dari *entry-level* hingga *flagship*.	Segmen pasar premium.

(Deskripsi Gambar: Dua screenshot berdampingan. Satu menunjukkan Home Screen Android (bisa pilih stock Android seperti Google Pixel, atau contoh UI populer seperti Samsung One UI untuk menunjukkan kustomisasi) dengan widget, berbagai icon, dan notification panel. Yang lainnya menunjukkan Home Screen iOS dengan icon aplikasi yang rapi dalam grid, widget khas iOS, dan Control Center atau Notification Center. Fokus pada perbedaan visual antara kedua OS.)

BAB 5. SISTEM OPERASI JARINGAN DAN TERDISTRIBUSI

Dalam lanskap komputasi modern, interkoneksi perangkat menjadi semakin vital. Sistem operasi tidak lagi beroperasi secara terisolasi pada satu mesin, melainkan harus mampu berinteraksi dan mengelola sumber daya yang tersebar di berbagai node dalam sebuah jaringan. Bab ini akan membahas dua konsep fundamental dalam domain ini: Sistem Operasi Jaringan (*Network Operating System*) dan Sistem Operasi Terdistribusi (*Distributed Operating System*). Kita akan mengeksplorasi perbedaan mendasar di antara keduanya, menyelami bagaimana sumber daya dikelola dalam lingkungan terdistribusi, serta meninjau contoh sistem operasi yang dirancang khusus untuk memenuhi kebutuhan komputasi jaringan dan terdistribusi, seperti Google Fuchsia dan Plan 9. Pemahaman terhadap arsitektur ini krusial untuk menghadapi tantangan dan peluang dalam era komputasi yang semakin terhubung.

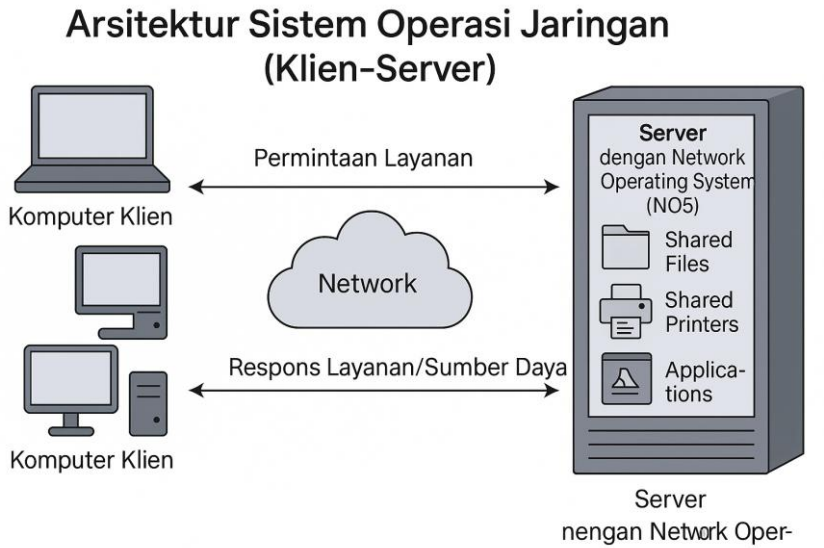
A. Sistem Operasi Jaringan (Network Operating System)

Sistem Operasi Jaringan (NOS) adalah jenis sistem operasi yang dirancang khusus untuk mendukung komputer pribadi, *workstation*, dan, yang paling utama, *server* agar dapat berfungsi dalam sebuah jaringan. Tujuan utama NOS adalah untuk memungkinkan berbagai komputer (klien) dalam jaringan untuk berbagi sumber daya perangkat keras dan perangkat lunak yang tersebar di seluruh *server* jaringan. NOS memfasilitasi komunikasi antar node dan menyediakan layanan jaringan yang esensial.

- Definisi dan Fungsi Utama: NOS adalah sistem operasi yang menjalankan *server* dan memungkinkan *client* untuk berbagi *file*, printer, aplikasi, dan sumber daya jaringan lainnya. NOS memiliki kemampuan untuk mengenali dan merespons permintaan dari berbagai pengguna di jaringan. Fungsi utamanya meliputi:
    - Manajemen Sumber Daya Jaringan: Mengontrol akses ke sumber daya bersama seperti *file server*, *print server*, dan *application server*. NOS memastikan bahwa beberapa pengguna dapat mengakses sumber daya ini secara bersamaan tanpa konflik.
    - Manajemen Pengguna dan Grup: Mengelola akun pengguna, otentikasi (memverifikasi identitas pengguna), dan otorisasi (menentukan hak akses pengguna terhadap sumber daya). NOS memungkinkan administrator untuk membuat grup pengguna dengan hak akses tertentu, menyederhanakan manajemen keamanan.
    - Keamanan Jaringan: Menerapkan kebijakan keamanan seperti *firewall*, sistem deteksi intrusi, dan kontrol akses berbasis peran (Role-Based Access Control/RBAC) untuk melindungi data dan sumber daya dari akses tidak sah.
    - Direktori Layanan: Menyediakan layanan direktori (misalnya, Active Directory di Windows Server, LDAP di Linux) yang menyimpan informasi tentang pengguna, komputer, dan sumber daya jaringan, memudahkan pencarian dan pengelolaan objek di jaringan.
    - Dukungan Protokol Jaringan: Membangun dan mengelola *network stack* yang mendukung berbagai protokol komunikasi (TCP/IP, UDP, DNS, DHCP, SMB/CIFS, NFS), memungkinkan interkonektivitas yang luas.

    - Manajemen Konfigurasi dan Pembaruan: Memungkinkan administrator untuk mengelola konfigurasi jaringan dari satu lokasi terpusat dan menyebarkan pembaruan atau *patch* ke *client* dalam jaringan.
- Arsitektur Klien-Server: NOS umumnya beroperasi dalam model arsitektur klien-server. Dalam model ini, *server* adalah komputer kuat yang menjalankan NOS dan menyediakan layanan, sementara *klien* adalah komputer pengguna yang mengakses layanan tersebut.
    - Server: Bertanggung jawab untuk menyimpan *file*, mengelola *database*, menjalankan aplikasi bisnis, dan menangani permintaan *client*.
    - Klien: Mengirimkan permintaan ke *server* dan menampilkan informasi yang diterima. Meskipun *klien* juga memiliki sistem operasinya sendiri (misalnya, Windows 11, macOS), mereka menggunakan layanan jaringan yang disediakan oleh NOS di *server*.
- Contoh Sistem Operasi Jaringan:
    - Windows Server: Seri sistem operasi *server* dari Microsoft (misalnya, Windows Server 2019, 2022). Sangat populer di lingkungan *enterprise* untuk Active Directory, *file sharing*, *web hosting* (IIS), dan aplikasi bisnis berbasis Windows.
    - Linux (misalnya, Ubuntu Server, Red Hat Enterprise Linux, CentOS): Distribusi Linux sangat dominan di lingkungan *server* untuk *web server* (Apache, Nginx), *database server* (MySQL, PostgreSQL), *file server* (NFS, Samba), dan layanan jaringan lainnya. Fleksibilitas, stabilitas, dan sifat *open source*-nya menjadikannya pilihan utama.

- Unix (misalnya, Solaris, HP-UX, AIX): Sistem operasi *enterprise* yang robust, meskipun penggunaannya telah menurun dibandingkan Linux.



(Deskripsi Gambar: Diagram yang menunjukkan beberapa "Komputer Klien" (misalnya Laptop, Desktop) yang terhubung ke sebuah "Jaringan" (misalnya ikon awan atau garis penghubung). Di sisi lain Jaringan, ada sebuah "Server" besar yang berlabel "Server dengan Network Operating System (NOS)". Tunjukkan panah dua arah antara Klien dan Server melalui Jaringan, dengan label "Permintaan Layanan" dari Klien ke Server dan "Respons Layanan/Sumber Daya" dari Server ke Klien. Di dalam Server, bisa ditunjukkan ikon atau label untuk "Shared Files", "Shared Printers", "Applications" untuk menjelaskan sumber daya yang dibagi.)

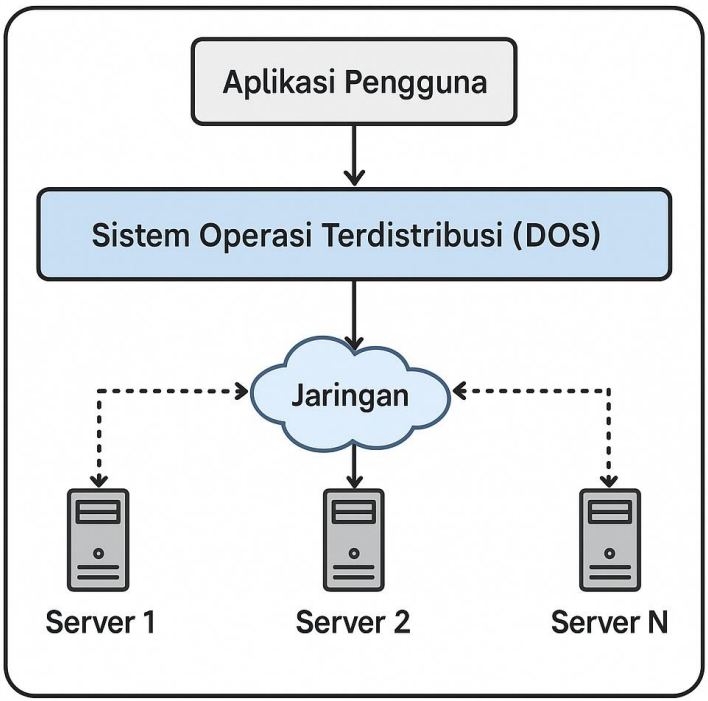
B. Sistem Operasi Terdistribusi (Distributed Operating System)

Berbeda dengan Sistem Operasi Jaringan yang membedakan secara jelas peran *klien* dan *server*, Sistem Operasi Terdistribusi (DOS) mengambil konsep manajemen jaringan ke tingkat yang lebih tinggi. DOS bertujuan untuk mengelola sekumpulan komputer yang terhubung dalam sebuah jaringan sebagai satu sistem komputasi tunggal yang kohesif. Dari sudut pandang pengguna, semua mesin ini berfungsi seperti satu mesin tunggal, menyembunyikan kompleksitas dari infrastruktur yang mendasarinya.

- Definisi dan Tujuan: Sistem Operasi Terdistribusi adalah kumpulan prosesor independen yang saling berkomunikasi melalui jaringan, dan dari perspektif pengguna, sistem ini tampak sebagai satu komputer tunggal yang terpadu. Tujuan utamanya adalah untuk menciptakan transparansi jaringan, di mana pengguna tidak perlu tahu di mana sumber daya atau proses sebenarnya berada atau dijalankan.
- Karakteristik Kunci DOS:
    - Transparansi (Transparency): Ini adalah karakteristik paling penting. DOS berusaha menyembunyikan sifat terdistribusi dari pengguna. Berbagai jenis transparansi meliputi:
        - Transparansi Lokasi: Pengguna tidak perlu tahu di mana sumber daya (file, printer, proses) berada secara fisik di jaringan.
        - Transparansi Akses: Pengguna dapat mengakses sumber daya di mana pun mereka berada di jaringan dengan cara yang seragam.
        - Transparansi Konkurensi: Banyak pengguna dapat berbagi sumber daya tanpa intervensi.

        - Transparansi Kegagalan: Sistem dapat terus berfungsi meskipun beberapa komponen mengalami kegagalan.
        - Transparansi Replikasi: Jika suatu sumber daya direplikasi untuk ketersediaan, pengguna tidak menyadarinya.
    - Skalabilitas: Kemampuan untuk dengan mudah menambah atau mengurangi node dalam sistem untuk mengakomodasi beban kerja yang bervariasi tanpa mengganggu operasi.
    - Konkurensi: Memungkinkan eksekusi paralel dari banyak proses di berbagai node, meningkatkan *throughput*.
    - Toleransi Kesalahan (Fault Tolerance): Dirancang untuk terus beroperasi meskipun ada kegagalan sebagian pada node tertentu, biasanya melalui replikasi data atau mekanisme *failover*.
    - Ketersediaan Tinggi (High Availability): Memastikan bahwa layanan dan sumber daya selalu tersedia bagi pengguna.
    - Keterbukaan (Openness): Kemampuan untuk dengan mudah memperluas atau memodifikasi sistem, mendukung protokol standar dan API yang memungkinkan komponen dari vendor berbeda untuk berinteraksi.
- Perbedaan NOS vs. DOS: Seringkali terjadi kebingungan antara NOS dan DOS. Perbedaan utamanya terletak pada tingkat abstraksi dan transparansi yang ditawarkan.
    - NOS: Pengguna sadar bahwa mereka mengakses sumber daya di *server* terpisah (misalnya, "\server\shared_folder"). NOS mengelola koneksi antara *klien* dan *server*.
    - DOS: Pengguna melihat semua sumber daya sebagai bagian dari satu sistem global (misalnya, mengakses *file* tanpa mengetahui di *server* fisik mana *file* itu disimpan). DOS

mengelola *pool* sumber daya yang terdistribusi seolah-olah itu adalah satu *pool* tunggal.



C. Manajemen Sumber Daya Terdistribusi

Manajemen sumber daya dalam sistem operasi terdistribusi jauh lebih kompleks daripada di sistem tunggal. DOS harus secara efisien mengelola proses, memori, dan sistem *file* yang tersebar di banyak node, sambil mempertahankan transparansi.

- Manajemen Proses Terdistribusi:
  - Penjadwalan Tugas: DOS harus menentukan node mana yang paling sesuai untuk menjalankan suatu proses atau tugas,

berdasarkan ketersediaan sumber daya (CPU, memori), beban jaringan, atau bahkan lokasi data.
    - Migrasi Proses: Kemampuan untuk memindahkan proses yang sedang berjalan dari satu node ke node lain (misalnya, untuk *load balancing* atau toleransi kesalahan).
    - Komunikasi Antarproses (IPC) Terdistribusi: Mekanisme untuk proses yang berjalan di node berbeda dapat saling berkomunikasi secara transparan (misalnya, melalui *Remote Procedure Call/RPC* atau *message passing*).
    - Sinkronisasi dan Koherensi: Memastikan bahwa peristiwa yang terjadi di node berbeda disinkronkan dengan benar (misalnya, menggunakan *logical clocks* atau *vector clocks*) dan bahwa data yang dibagikan tetap konsisten di seluruh sistem.
- Manajemen Memori Terdistribusi:
    - Memori Bersama Terdistribusi (Distributed Shared Memory/DSM): Sebuah abstraksi yang memungkinkan proses di node berbeda untuk mengakses ruang alamat virtual bersama, seolah-olah mereka berada di satu mesin. DOS mengelola replikasi dan koherensi halaman memori di antara node.
    - Penukaran dan Paging Terdistribusi: Mekanisme untuk mengelola penggunaan memori di seluruh sistem, termasuk *paging* data ke atau dari penyimpanan terdistribusi.
- Sistem File Terdistribusi (Distributed File System/DFS): DFS memungkinkan pengguna untuk mengakses *file* yang disimpan di berbagai *server* jaringan seolah-olah *file* tersebut berada di penyimpanan lokal. DFS bertanggung jawab untuk:

- Transparansi Lokasi dan Akses: Pengguna tidak perlu tahu di *server* mana *file* disimpan.
- Replikasi dan Caching: Mereplikasi *file* di beberapa lokasi atau menyimpan *cache* di *client* untuk meningkatkan ketersediaan dan kinerja.
- Koherensi Data: Memastikan bahwa semua salinan *file* yang direplikasi tetap konsisten ketika ada perubahan.
- Toleransi Kegagalan: Jika satu *server file* gagal, *file* masih dapat diakses dari replika lain.
- Contoh DFS termasuk NFS (Network File System), SMB/CIFS (Server Message Block/Common Internet File System), dan Google File System (GFS) yang digunakan secara internal oleh Google.

- Algoritma Konsensus: Dalam sistem terdistribusi, mencapai konsensus di antara node tentang suatu nilai atau status (misalnya, siapa pemimpin, status transaksi) adalah tantangan besar. Algoritma seperti Paxos atau Raft digunakan untuk memastikan bahwa semua node setuju pada urutan operasi atau nilai tertentu, bahkan jika ada kegagalan node atau jaringan. Ini krusial untuk menjaga konsistensi data dan keandalan sistem.

D. Contoh OS Terdistribusi (Google Fuchsia, Plan 9)

Meskipun banyak sistem modern mengadopsi prinsip terdistribusi (misalnya, *cloud OS* yang mendasari layanan *cloud*), ada beberapa SO yang secara eksplisit dirancang dari awal sebagai sistem operasi terdistribusi.

- Google Fuchsia:
  - Filosofi: Fuchsia adalah sistem operasi *open source* yang sedang dikembangkan oleh Google. Berbeda dengan Android

atau Chrome OS yang berbasis kernel Linux, Fuchsia dibangun di atas mikrokernel baru bernama Zircon. Desain mikrokernel ini bertujuan untuk mencapai skalabilitas yang lebih baik, keamanan yang lebih kuat, dan kemampuan untuk beradaptasi dengan berbagai jenis perangkat keras, dari perangkat IoT yang sangat kecil hingga *smartphone*, tablet, *laptop*, dan bahkan *desktop*.

- Karakteristik Terdistribusi: Meskipun bukan DOS tradisional, arsitektur modular Zircon dan fokusnya pada layanan memungkinkan komponen sistem berinteraksi secara terdistribusi. Tujuan jangka panjang Fuchsia adalah untuk menjadi SO yang *future-proof* untuk dunia yang semakin terhubung dan terdistribusi, di mana perangkat dapat bekerja sama secara kohesif tanpa *user* perlu tahu di mana komputasi terjadi. Ini mendukung konsep *Ambient Computing* Google.
- Penggunaan Saat Ini: Masih dalam tahap pengembangan aktif, meskipun telah diterapkan pada beberapa perangkat *smart home* Google (misalnya, Google Nest Hub generasi ke-2).

- Plan 9 from Bell Labs:
  - Filosofi: Dikembangkan pada akhir 1980-an di Bell Labs (oleh beberapa orang yang juga mengembangkan Unix), Plan 9 adalah eksperimen radikal dalam sistem operasi terdistribusi. Filosofi intinya adalah "semuanya adalah *file*". Setiap sumber daya dalam sistem (proses, jaringan, perangkat keras, bahkan GUI) direpresentasikan sebagai *file* dalam sistem *file* hierarkis. Pengguna mengakses dan mengelola sumber daya ini melalui operasi *file* standar.
  - Karakteristik Terdistribusi: Plan 9 secara intrinsik terdistribusi. Pengguna dapat "me-*mount*" sistem *file* dari

server mana pun ke *client* lokal, mengakses sumber daya jarak jauh secara transparan. Ini memungkinkan komputasi *grid* yang mulus dan berbagi sumber daya yang mudah di seluruh jaringan.

- Fitur Utama:
  - Protokol 9P: Protokol jaringan ringan yang digunakan untuk semua komunikasi di Plan 9, yang didasarkan pada konsep *file* ini.
  - *Per-User File System Namespace*: Setiap pengguna memiliki *namespace* sistem *file* unik yang dapat mereka atur sendiri, menggabungkan sumber daya lokal dan jarak jauh.
- Penggunaan: Meskipun tidak pernah mencapai popularitas seperti Unix atau Linux, Plan 9 telah menjadi inspirasi penting dalam penelitian sistem operasi terdistribusi dan *cloud computing*. Konsepnya tentang "semuanya adalah *file*" dan *per-user namespace* telah mempengaruhi desain sistem modern lainnya.

Google Fuchsia

Plan 9 from
Bell Labs

BAB 6: KEAMANAN PADA SISTEM OPERASI MODERN

Dalam lanskap komputasi yang semakin terhubung dan kompleks, keamanan telah berevolusi dari sekadar fitur tambahan menjadi pilar fundamental dalam desain dan operasional sistem operasi modern. Ancaman siber yang terus berkembang, mulai dari *malware*, *ransomware*, hingga serangan tingkat tinggi, menuntut sistem operasi untuk tidak hanya mengelola sumber daya, tetapi juga melindungi integritas, kerahasiaan, dan ketersediaan data serta sistem itu sendiri. Bab ini akan menyelami berbagai model keamanan modern yang diimplementasikan dalam SO, termasuk konsep *Mandatory Access Control* (MAC) dengan studi kasus SELinux dan AppArmor. Selanjutnya, akan dibahas mekanisme vital seperti isolasi proses dan *sandboxing*, pentingnya enkripsi dan perlindungan data, serta strategi *update* keamanan dan *patch management* yang proaktif. Pemahaman mendalam tentang konsep-konsep ini sangat krusial bagi siapa pun yang berinteraksi dengan teknologi informasi di era digital ini.

A. Model Keamanan Modern (Mandatory Access Control, SELinux, AppArmor)

Keamanan dalam sistem operasi tradisional seringkali bergantung pada model Discretionary Access Control (DAC). Dalam DAC, pemilik sumber daya (misalnya, *file* atau proses) memiliki kebebasan untuk menentukan siapa yang dapat mengakses sumber daya tersebut dan dengan izin apa (baca, tulis, eksekusi). Meskipun fleksibel, DAC memiliki kelemahan serius: jika akun pengguna atau aplikasi terkompromi, penyerang dapat mengubah izin untuk

mengakses dan merusak sumber daya lain. Ini membuat sistem rentan terhadap serangan *privilege escalation* atau penyebaran *malware*.

Sebagai respons, sistem operasi modern telah mengadopsi model keamanan yang lebih kuat, salah satunya adalah Mandatory Access Control (MAC).

- Mandatory Access Control (MAC): Filosofi Keamanan Pusat
  Berbeda dengan DAC, dalam model MAC, kebijakan akses ditentukan dan diberlakukan oleh sistem itu sendiri (kernel SO), bukan oleh pemilik sumber daya atau pengguna. Administrator sistem (atau kebijakan keamanan yang telah ditetapkan) menentukan *policy* keamanan yang mengklasifikasikan setiap subjek (proses, pengguna) dan objek (file, port, perangkat) dengan label keamanan (*security label*). Akses hanya diberikan jika label keamanan subjek memenuhi persyaratan label keamanan objek sesuai dengan *policy* yang telah ditentukan. Kebijakan ini tidak dapat diubah oleh pengguna atau aplikasi, bahkan jika mereka memiliki *privilege* administratif.

  Kelebihan MAC:

  - Isolasi yang Kuat: Mencegah aplikasi atau pengguna yang terkompromi untuk mengakses atau merusak bagian lain dari sistem, bahkan jika mereka mendapatkan *root privilege*.
  - Konsistensi Kebijakan: Kebijakan akses diterapkan secara seragam di seluruh sistem, mengurangi risiko *human error* atau konfigurasi yang salah.
  - Pertahanan Mendalam: Menjadi lapisan pertahanan tambahan di luar DAC, terutama efektif melawan *zero-day exploits* dan *malware* yang mencoba menaikkan *privilege*.

  Kekurangan MAC (dan Tantangannya):

  - Kompleksitas Konfigurasi: Menerapkan dan mengelola kebijakan MAC bisa sangat kompleks dan memerlukan pemahaman mendalam tentang bagaimana sistem beroperasi.

- Dapat Mengganggu Fungsionalitas: Konfigurasi yang terlalu ketat dapat memblokir operasi yang sah, menyebabkan aplikasi tidak berfungsi.

- SELinux (Security-Enhanced Linux): Implementasi MAC yang Robust SELinux adalah mekanisme keamanan *Mandatory Access Control* yang diimplementasikan sebagai modul kernel Linux. Ini dikembangkan oleh National Security Agency (NSA) Amerika Serikat dan dirilis sebagai *open source*. SELinux bekerja dengan menambahkan atribut keamanan (*security context*) ke setiap objek dalam sistem (*file*, proses, *socket*, dll.) dan setiap subjek (proses). Kernel SELinux kemudian memeriksa *security context* ini terhadap *policy* yang dimuat untuk menentukan apakah suatu operasi diizinkan atau tidak.

  Cara Kerja SELinux:

  1. Setiap objek (misalnya, /var/www/html/index.php) memiliki *security context* (misalnya, system_u:object_r:httpd_sys_content_t:s0).
  2. Setiap proses (misalnya, proses httpd) memiliki *security context* (misalnya, system_u:system_r:httpd_t:s0).
  3. Ketika proses httpd mencoba mengakses index.php, kernel SELinux berkonsultasi dengan *policy matrix* untuk melihat apakah proses dengan *context* httpd_t diizinkan untuk membaca *file* dengan *context* httpd_sys_content_t.
  4. Jika tidak diizinkan, akses akan ditolak, bahkan jika izin DAC (misalnya, *permission Unix*) mengizinkannya.

  SELinux dapat beroperasi dalam mode *enforcing* (menegakkan kebijakan dan memblokir akses) atau *permissive* (mencatat pelanggaran tanpa memblokir akses, berguna untuk *troubleshooting*). Distribusi Linux seperti Red Hat Enterprise

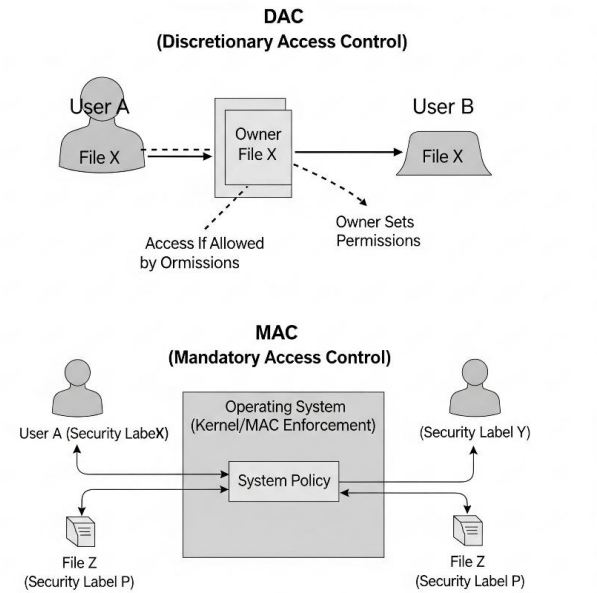
Linux (RHEL), Fedora, dan CentOS menggunakan SELinux secara *default*.

- AppArmor: Alternatif MAC yang Lebih Sederhana AppArmor (Application Armor) adalah mekanisme *Mandatory Access Control* lain yang tersedia di Linux, dikembangkan oleh Novell. Berbeda dengan SELinux yang berorientasi pada label keamanan yang sangat granular, AppArmor lebih berorientasi pada jalur (*path*) dan berbasis profil. Administrator membuat profil keamanan untuk aplikasi tertentu yang menentukan sumber daya (file, jaringan, kapabilitas kernel) apa yang boleh dan tidak boleh diakses oleh aplikasi tersebut.

  Cara Kerja AppArmor:

  1. Administrator mendefinisikan "profil" untuk suatu aplikasi (misalnya, *web server* Apache).
  2. Profil ini berisi daftar eksplisit tentang *file* apa yang boleh dibaca atau ditulis, *port* jaringan apa yang boleh dibuka, dan kemampuan sistem apa yang boleh digunakan oleh aplikasi tersebut.
  3. Kernel AppArmor akan memblokir setiap upaya akses yang dilakukan oleh aplikasi yang melanggar profilnya.

  AppArmor sering dianggap lebih mudah untuk dikonfigurasi dan dikelola dibandingkan SELinux, karena sifatnya yang lebih ringkas dan berbasis *path*. Distribusi Linux seperti Ubuntu, openSUSE, dan Debian menggunakan AppArmor secara *default*.



1. Diagram DAC: Tunjukkan "Pengguna A" dan "Pengguna B". Pengguna A memiliki "File X". Pengguna A memiliki panah ke File X dengan label "Pemilik Mengatur Izin". Pengguna B memiliki panah putus-putus ke File X dengan label "Akses Jika Diizinkan Pemilik". Tekankan bahwa Pengguna A bisa mengubah izin sesuka hati.
2. Diagram MAC: Tunjukkan "Pengguna A (Label Keamanan X)" dan "Pengguna B (Label Keamanan Y)". Tunjukkan "File Z (Label Keamanan P)". Sebuah kotak besar "Sistem Operasi (Kernel/MAC Enforcement)" berada di antara Pengguna dan File. Panah dari Pengguna ke Sistem Operasi, dan dari Sistem Operasi ke File. Di tengah Sistem Operasi, tulis "Kebijakan Sistem". Tekankan bahwa Sistem Operasi yang menentukan apakah Pengguna dapat mengakses File berdasarkan Label Keamanan dan Kebijakan, bukan pemilik File.)*

B. Isolasi Proses dan Sandbox

Salah satu prinsip keamanan fundamental dalam sistem operasi modern adalah isolasi. Tujuannya adalah untuk membatasi dampak dari *bug* atau *malware* pada satu aplikasi atau proses agar tidak menyebar dan merusak seluruh sistem. Dua mekanisme utama untuk mencapai isolasi ini adalah isolasi proses dan *sandboxing*.

- Isolasi Proses: Setiap proses (misalnya, sebuah aplikasi yang sedang berjalan) dalam sistem operasi modern dialokasikan dengan ruang memori virtualnya sendiri yang terpisah dari proses lain. Ini berarti satu proses tidak dapat secara langsung membaca atau menulis ke ruang memori proses lain, kecuali melalui mekanisme komunikasi antarproses (IPC) yang dikontrol.
  Manfaat Isolasi Proses:
  - Stabilitas Sistem: Jika satu aplikasi mengalami *crash* karena *bug* atau kehabisan memori, itu tidak akan merusak proses lain atau kernel sistem operasi. Hanya aplikasi tersebut yang berhenti berfungsi.
  - Keamanan: Mencegah satu aplikasi berbahaya untuk memata-matai atau memanipulasi data dari aplikasi lain. Ini adalah fondasi dari *multi-tasking* yang aman.
  - Manajemen Sumber Daya: Memungkinkan SO untuk melacak dan mengelola sumber daya (CPU, memori) yang digunakan oleh setiap proses secara individu.

  Mekanisme inti di balik isolasi proses adalah manajemen memori virtual (MMU - *Memory Management Unit*) pada prosesor, yang menerjemahkan alamat memori virtual yang dilihat oleh proses menjadi alamat memori fisik yang sebenarnya, dan memberlakukan batasan akses.
- Sandbox (Lingkungan Terpasir): *Sandbox* adalah mekanisme keamanan yang lebih ketat dari isolasi proses, menciptakan

lingkungan eksekusi yang sangat terisolasi untuk program. Lingkungan ini memiliki batasan yang ketat mengenai sumber daya apa yang dapat diakses oleh program, termasuk *file*, *network connection*, dan *hardware* tertentu. Aplikasi di dalam *sandbox* hanya dapat berinteraksi dengan sumber daya di luar *sandbox* melalui API yang sangat terkontrol dan dengan izin eksplisit.

Cara Kerja Sandbox: Sebuah *sandbox* bekerja dengan menempatkan program di dalam "kurungan" virtual yang membatasi aksesnya ke sumber daya di luar batas yang diizinkan. Ini sering dilakukan melalui:

- Virtualisasi Level Aplikasi: Menggunakan teknologi seperti kontainer (*containers*) atau mesin virtual ringan.
- Mekanisme Kernel: SO membatasi panggilan sistem (*system calls*) yang dapat dilakukan oleh proses di dalam *sandbox*.
- Kebijakan Keamanan: Aturan yang telah ditetapkan mengenai *file* mana yang boleh diakses, *port* jaringan mana yang boleh digunakan, atau perangkat *hardware* mana yang dapat diakses.

Penerapan Sandbox dalam SO Modern:

- Browser Web: Setiap tab atau *extension* di *browser* modern (Chrome, Firefox, Edge) sering berjalan di *sandbox* terpisah. Jika ada situs web berbahaya yang mencoba mengeksploitasi *browser*, kerusakannya terbatas pada *sandbox* tab tersebut, tidak dapat merusak seluruh sistem operasi.
- Aplikasi Mobile: Baik Android maupun iOS sangat mengandalkan *sandboxing* untuk setiap aplikasi. Aplikasi tidak dapat mengakses *file* atau data aplikasi lain tanpa izin eksplisit yang diberikan oleh pengguna. Misalnya, aplikasi *game* tidak bisa membaca kontak Anda tanpa persetujuan Anda.
- Aplikasi Desktop: Beberapa aplikasi desktop modern (misalnya, aplikasi yang diinstal dari Windows Store, atau

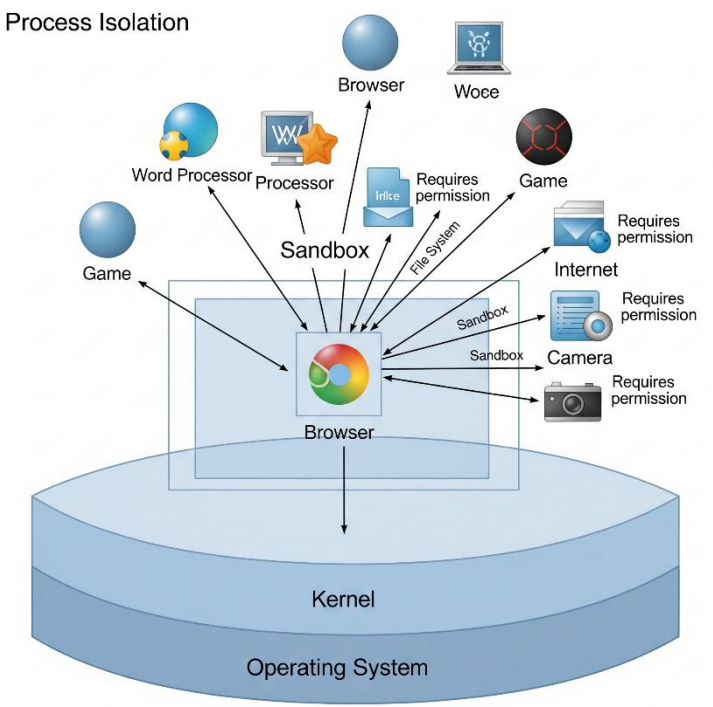
aplikasi di macOS App Store) juga di-*sandbox* untuk meningkatkan keamanan.

- Eksekusi Kode yang Tidak Dipercaya: *Sandbox* digunakan untuk menjalankan kode yang tidak dipercaya (misalnya, *plugin* pihak ketiga, skrip) di lingkungan yang aman, mencegahnya melakukan tindakan berbahaya

Manfaat Sandbox:

- Perlindungan Terhadap Ancaman: Membatasi dampak *malware*, *exploit*, atau kode berbahaya lainnya.
- Pengurangan Permukaan Serangan: Mengurangi area sistem yang dapat diakses oleh kode yang tidak dipercaya.

- Perlindungan Privasi: Mencegah aplikasi mengakses data pribadi tanpa izin pengguna.



C. Enkripsi dan Perlindungan Data

Perlindungan data adalah aspek krusial keamanan sistem operasi modern. Ini melibatkan penggunaan kriptografi untuk menjaga kerahasiaan dan integritas data, baik saat data tersebut disimpan (data *at rest*) maupun saat sedang ditransfer (data *in transit*).

- Enkripsi Data At Rest (Penyimpanan): Enkripsi data *at rest* adalah praktik mengkodekan data yang disimpan di perangkat penyimpanan (*hard drive*, SSD, *flash drive*) sehingga tidak dapat dibaca oleh pihak yang tidak sah. Bahkan jika perangkat dicuri atau diakses secara fisik, data tidak dapat diakses tanpa kunci dekripsi yang benar.
    - Enkripsi Disk Penuh (*Full-Disk Encryption* / FDE): Ini adalah metode paling umum, di mana seluruh *drive* atau partisi dienkripsi. SO mengelola proses enkripsi/dekripsi secara transparan bagi pengguna setelah otentikasi awal (misalnya, memasukkan *password boot*).

      Contoh:
        - BitLocker (Windows): Fitur FDE bawaan pada edisi Windows Pro dan Enterprise.
        - FileVault (macOS): Fitur FDE bawaan untuk perangkat Apple.
        - LUKS (Linux Unified Key Setup) / dm-crypt (Linux): Standar untuk enkripsi *disk* di sebagian besar distribusi Linux.
        - Android/iOS Default Encryption: Sebagian besar perangkat *mobile* modern mengenkripsi seluruh penyimpanan secara *default* dan terikat pada otentikasi *fingerprint* atau *passcode* pengguna.

    - Enkripsi File/Folder: Mengenkripsi *file* atau *folder* tertentu, bukan seluruh *drive*. Ini berguna untuk melindungi data sensitif di lingkungan *multi-user* atau *share*. Contoh: *Encrypting File System* (EFS) di Windows.
- Enkripsi Data In Transit (Transfer Data): Enkripsi data *in transit* melindungi data saat ditransfer melalui jaringan (misalnya, internet, jaringan lokal). Tujuannya adalah untuk mencegah *eavesdropping* (menguping) atau *man-in-the-middle attacks*.
    - TLS/SSL (Transport Layer Security/Secure Sockets Layer): Protokol kriptografi yang digunakan untuk mengamankan komunikasi melalui jaringan komputer. Digunakan secara luas untuk *web Browse* (HTTPS), *email* (SMTPS, IMAPS), dan VPN. OS modern menyediakan *stack* jaringan yang mendukung TLS/SSL secara *native*.
    - VPN (Virtual Private Network): Membuat "terowongan" terenkripsi melalui jaringan publik, memungkinkan pengguna untuk mengirim dan menerima data secara aman seolah-olah perangkat mereka terhubung langsung ke jaringan privat. OS menyediakan klien VPN bawaan atau mendukung *software* VPN pihak ketiga.
    - SSH (Secure Shell): Protokol untuk akses jarak jauh yang aman ke *server* dan untuk transfer *file* yang aman. SO modern mendukung klien dan *server* SSH.
- Perlindungan Integritas Data: Selain kerahasiaan (enkripsi), perlindungan data juga mencakup integritas, yaitu memastikan data tidak dimodifikasi secara tidak sah. Ini sering dicapai melalui:
    - Hashing dan Digital Signatures: Menggunakan fungsi *hash* kriptografi untuk menghasilkan nilai unik dari data, yang

kemudian dapat diverifikasi menggunakan tanda tangan digital untuk memastikan data tidak berubah. Digunakan dalam pembaruan perangkat lunak, verifikasi *file*, dan *boot* aman.

- Journaling File Systems: Sistem *file* modern (misalnya, NTFS, ext4, APFS) menggunakan *journaling* untuk mencatat perubahan yang tertunda ke *disk*. Ini membantu memulihkan sistem *file* ke keadaan konsisten setelah kegagalan daya atau *crash* sistem, mencegah korupsi data.

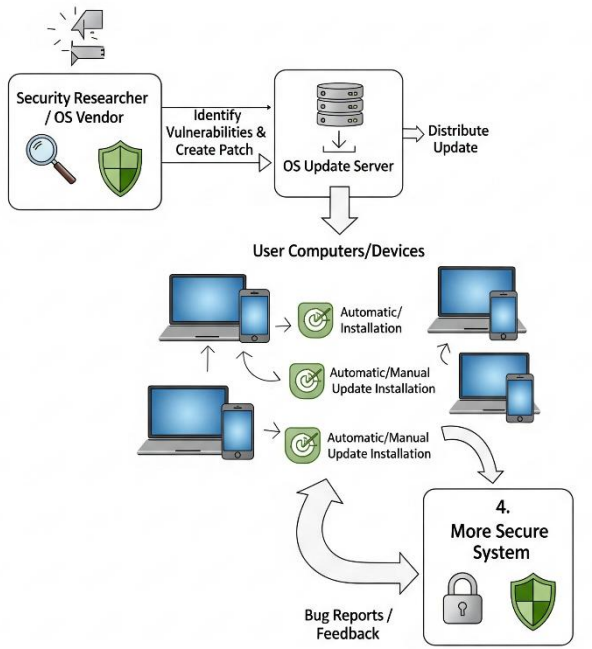
D. Update Keamanan dan Patch Management

Salah satu lini pertahanan paling efektif dalam keamanan sistem operasi modern adalah sistem *update* dan *patch management* yang proaktif. Kerentanan (*vulnerability*) keamanan baru ditemukan secara terus-menerus, dan *malware* serta penyerang dengan cepat mengeksploitasi celah ini. Oleh karena itu, kemampuan sistem operasi untuk secara rutin menerima dan menerapkan pembaruan sangat penting.

- Pentingnya Pembaruan Keamanan:
  - Menambal Kerentanan (*Patching Vulnerabilities*): Pengembang SO secara aktif mencari dan memperbaiki *bug* atau celah keamanan (*exploit*) yang dapat dimanfaatkan oleh penyerang. Pembaruan ini, sering disebut *patch*, dirilis untuk menutup celah tersebut.
  - Melawan Ancaman Baru: Pembaruan juga dapat mencakup peningkatan mekanisme keamanan untuk menghadapi jenis serangan baru atau *malware* yang berkembang.
  - Kepatuhan: Banyak standar keamanan dan regulasi (misalnya, ISO 27001, HIPAA, GDPR) mensyaratkan *patch management* yang efektif sebagai bagian dari upaya kepatuhan.

- Mekanisme Pembaruan Otomatis OS: Sistem operasi modern telah mengotomatiskan proses *update* sebisa mungkin untuk memastikan pengguna dan administrator tetap terlindungi.
    - Windows Update: Windows secara teratur merilis *patch* keamanan dan pembaruan fitur. Windows 10 dan 11 secara *default* mengaktifkan pembaruan otomatis, seringkali dengan kemampuan untuk mengunduh di latar belakang dan menginstal di luar jam kerja.
    - Software Update (macOS): Apple juga menyediakan pembaruan sistem secara rutin yang mencakup *patch* keamanan dan peningkatan fitur, dengan opsi untuk instalasi otomatis.
    - Package Managers (Linux): Distribusi Linux menggunakan *package manager* (misalnya, apt di Debian/Ubuntu, dnf di Fedora/RHEL) untuk mengelola perangkat lunak dan pembaruan sistem. Administrator atau pengguna dapat menjalankan perintah untuk mengunduh dan menginstal *patch* keamanan. Banyak sistem Linux *server* dikonfigurasi untuk pembaruan keamanan otomatis.
    - Pembaruan OTA (Over-The-Air) pada Mobile OS: Android dan iOS secara rutin mengirimkan pembaruan OTA ke perangkat mobile. Pembaruan ini mencakup *patch* keamanan, perbaikan *bug*, dan fitur baru.
        - Tantangan Fragmentasi (Android): Meskipun Google merilis *patch* bulanan, penyebaran *patch* ke perangkat pengguna akhir seringkali terhambat oleh produsen perangkat dan operator, yang perlu menyesuaikan dan menguji pembaruan sebelum mendistribusikannya. Ini menciptakan "kesenjangan *patch*".

        - Keunggulan Kontrol (iOS): Apple memiliki kontrol vertikal, sehingga *patch* keamanan iOS dapat menjangkau sebagian besar perangkat yang didukung dengan sangat cepat dan seragam.
- Patch Management: *Patch management* adalah proses sistematis untuk mengidentifikasi, memperoleh, menguji, dan menyebarkan *patch* perangkat lunak. Dalam lingkungan *enterprise*, ini melibatkan strategi yang lebih kompleks untuk memastikan pembaruan tidak mengganggu aplikasi kritis dan dikelola secara terpusat.
    - Pengujian: Sebelum *patch* diterapkan di lingkungan produksi, mereka sering diuji di lingkungan *staging* untuk memastikan tidak ada konflik atau regresi yang muncul.
    - Rollback Options: SO modern sering menyediakan kemampuan untuk *rollback* ke versi sebelumnya jika pembaruan menyebabkan masalah serius.
    - Vulnerability Disclosure dan CVE: Kerentanan keamanan seringkali dicatat dalam *Common Vulnerabilities and Exposures* (CVE) *database*, memberikan standar untuk mengidentifikasi dan melacak kerentanan yang telah ditambal.



1. Kotak "Peneliti Keamanan / Vendor SO": Dengan ikon kaca pembesar (mencari kerentanan) dan ikon perisai (mengembangkan *patch*). Panah keluar "Identifikasi Kerentanan & Buat Patch".
2. Panah ini mengarah ke Kotak "Server Pembaruan OS" (ikon *server* dengan panah *download*). Panah keluar "Distribusi Update".
3. Panah ini mengarah ke beberapa "Komputer/Perangkat Pengguna" (ikon Laptop, Smartphone). Di setiap perangkat, tunjukkan ikon "Instalasi Otomatis/Manual Update".
4. Hasil akhirnya adalah "Sistem yang Lebih Aman" (ikon gembok terkunci atau perisai hijau). Bisa tambahkan panah melingkar dari "Perangkat Pengguna" kembali ke "Peneliti/Vendor SO" dengan label "Laporan Bug / Feedback".)*

BAB 7: KONTROL VERSI, PEMBARUAN, DAN AUTOMASI

Pengembangan dan pemeliharaan sistem operasi modern adalah sebuah proses yang kompleks dan dinamis. Dengan jutaan baris kode dan komponen yang saling terkait, diperlukan metodologi yang canggih untuk mengelola perubahan, memastikan kualitas, dan mendistribusikan pembaruan secara efisien. Bab ini akan membahas tiga pilar penting dalam siklus hidup sistem operasi modern: *Continuous Integration* dan *Continuous Deployment* (CI/CD) sebagai praktik pengembangan perangkat lunak yang revolusioner; bagaimana sistem operasi mengelola pembaruan otomatis untuk menjaga keamanan dan fungsionalitas; serta pentingnya manajemen versi kernel dan komponen OS untuk stabilitas dan evolusi sistem. Pemahaman tentang otomatisasi dan praktik rekayasa perangkat lunak ini akan memberikan wawasan tentang bagaimana SO tetap relevan dan aman di tengah perkembangan teknologi yang cepat.

A. Continuous Integration dan Continuous Deployment (CI/CD)

Continuous Integration (CI) dan *Continuous Deployment* (CD) adalah praktik rekayasa perangkat lunak yang telah merevolusi cara pengembangan dan pengiriman *software*, termasuk sistem operasi dan komponennya. Tujuan utama CI/CD adalah untuk mempercepat siklus *release*, meningkatkan kualitas kode, dan mengurangi risiko kesalahan melalui otomatisasi yang ekstensif.

- Continuous Integration (CI): Integrasi Kode yang Berkelanjutan
  CI adalah praktik di mana pengembang secara teratur

mengintegrasikan perubahan kode mereka ke dalam repositori pusat (*mainline* atau *main branch*) berkali-kali dalam sehari. Setiap kali kode baru diintegrasikan, proses otomatis akan:

- Kompilasi Otomatis: Kode sumber dikompilasi untuk memastikan tidak ada kesalahan sintaksis atau dependensi yang hilang.
- Pengujian Otomatis: Berbagai jenis tes (misalnya, *unit tests*, *integration tests*, *static code analysis*) dijalankan secara otomatis untuk mendeteksi *bug* atau regresi sejak dini. Jika ada tes yang gagal, pengembang akan segera diberitahu.
- Feedback Cepat: Pengembang mendapatkan *feedback* instan mengenai kualitas dan fungsionalitas kode mereka. Ini memungkinkan masalah diidentifikasi dan diperbaiki saat masih kecil, jauh sebelum menjadi *bug* besar yang sulit dilacak.

Manfaat CI bagi Pengembangan OS:

- Deteksi Bug Dini: Mencegah integrasi kode yang rusak ke dalam *mainline*.
- Kualitas Kode yang Konsisten: Memastikan bahwa semua kontribusi memenuhi standar kualitas.
- Kolaborasi Efisien: Memungkinkan banyak pengembang untuk bekerja pada basis kode yang sama tanpa saling mengganggu.
- Pengurangan "Integration Hell": Menghindari masalah besar yang muncul ketika perubahan kode diintegrasikan sekaligus setelah periode pengembangan yang panjang.

- Continuous Deployment (CD): Pengiriman Otomatis ke Produksi CD adalah langkah selanjutnya setelah CI. Ini adalah praktik di mana setiap perubahan kode yang berhasil melewati semua tahap pengujian otomatis di CI akan secara otomatis dikirim (*deployed*)

ke lingkungan produksi atau *staging*. Tujuannya adalah untuk memastikan bahwa perangkat lunak selalu dalam kondisi yang dapat di-*deploy* setiap saat.

Continuous Delivery vs. Continuous Deployment: Perlu dicatat perbedaan antara *Continuous Delivery* dan *Continuous Deployment*.

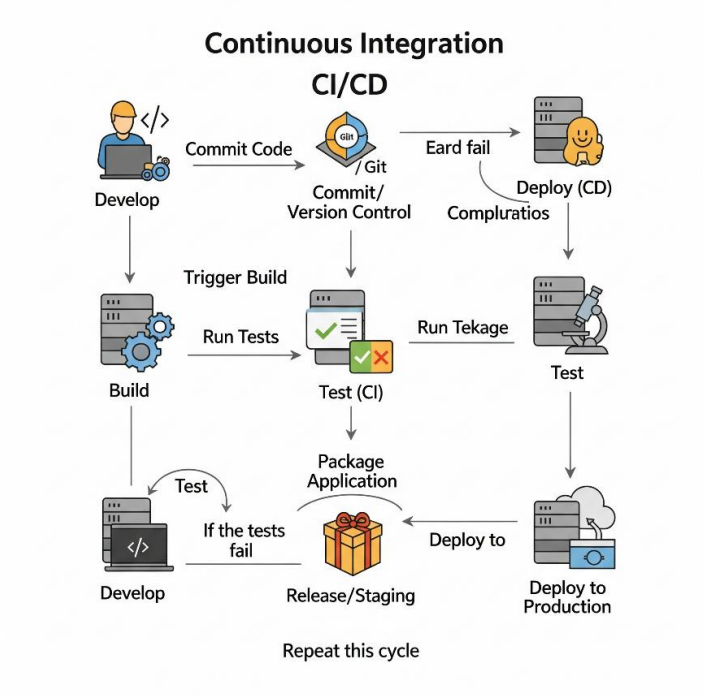
- *Continuous Delivery:* Mengotomatiskan semua langkah hingga tahap *deployment* ke produksi, tetapi membutuhkan persetujuan manual untuk *actual release* ke pengguna akhir.
- *Continuous Deployment:* Mengotomatiskan seluruh proses dari komit kode hingga *deployment* ke produksi tanpa intervensi manual, asalkan semua tes otomatis berhasil.

Manfaat CD bagi Sistem Operasi:

- Siklus Rilis Cepat: Pembaruan dan *patch* keamanan dapat didistribusikan dengan sangat cepat setelah dikembangkan.
- Respon Cepat terhadap Ancaman: Kerentanan keamanan dapat ditambal dan didistribusikan ke pengguna dalam hitungan jam atau hari, bukan minggu atau bulan.
- Inovasi Berkelanjutan: Fitur baru dapat di-*roll out* secara bertahap dan cepat, memungkinkan *developer* untuk bereksperimen dan mendapatkan *feedback* lebih cepat.
- Mengurangi Human Error: Otomatisasi menghilangkan banyak *human error* yang bisa terjadi dalam proses *deployment* manual.

Meskipun *Continuous Deployment* penuh pada OS *consumer* (seperti Windows atau macOS) mungkin jarang dilakukan karena risikonya, prinsip-prinsip CD sangat diterapkan pada komponen-komponen tertentu atau pada SO di lingkungan *cloud* dan *server* yang *highly-controlled*. Contohnya, Google

sering menggunakan CD untuk layanan internal mereka dan beberapa *server-side components* Android.



1. Develop: (Ikon pengembang mengetik kode). Panah "Commit Code".
2. Commit / Version Control: (Ikon Git/repositori kode). Panah "Trigger Build".
3. Build: (Ikon *server* dengan roda gigi/ikon kompilasi). Panah "Run Tests".
4. Test (CI): (Ikon *server* dengan tanda centang/X, atau ikon mikroskop). Panah "Package Application".
5. Release/Package: (Ikon kotak hadiah/paket). Panah "Deploy to Staging".
6. Deploy (CD): (Ikon *server* yang me-*deploy* ke *cloud* atau mesin). Panah "Deploy to Production".

7. Ulangi siklus ini. Tambahkan panah *feedback* dari "Test" kembali ke "Develop" jika gagal.)*

B. Sistem Pembaruan Otomatis OS

Pembaruan perangkat lunak, terutama untuk sistem operasi, sangat penting untuk menjaga keamanan, stabilitas, dan fungsionalitas. Sistem operasi modern telah mengembangkan mekanisme pembaruan otomatis yang canggih untuk menyederhanakan proses ini bagi pengguna akhir dan memastikan sistem tetap terlindungi dari ancaman baru.

- Pentingnya Pembaruan Otomatis:
  - Keamanan Kritis: Kerentanan keamanan baru ditemukan setiap hari. Pembaruan otomatis memastikan bahwa *patch* keamanan kritikal dapat diterapkan dengan cepat tanpa intervensi pengguna, menutup celah eksploitasi.
  - Perbaikan Bug: Mengatasi *bug* dan masalah kinerja yang ditemukan setelah *release* awal.
  - Fitur Baru dan Peningkatan Performa: Memperkenalkan fungsionalitas baru, kompatibilitas *hardware* yang lebih baik, dan optimasi kinerja.
  - Kepatuhan: Banyak lingkungan *enterprise* dan regulasi memerlukan sistem untuk selalu diperbarui.
- Mekanisme Pembaruan pada Berbagai OS Modern:
  - Windows Update: Ini adalah layanan bawaan Microsoft untuk mendistribusikan pembaruan ke sistem Windows. Sejak Windows 10, pembaruan otomatis diaktifkan secara *default* dan sangat sulit untuk dimatikan sepenuhnya pada edisi *consumer*. Pembaruan dibagi menjadi:

        - *Quality Updates (Cumulative Updates):* Pembaruan bulanan yang mencakup *patch* keamanan, perbaikan *bug*, dan peningkatan keandalan.
        - *Feature Updates:* Pembaruan besar yang dirilis satu atau dua kali setahun, membawa fitur baru, perubahan antarmuka, dan peningkatan signifikan. Windows Update seringkali mengunduh pembaruan di latar belakang dan memerlukan *restart* perangkat. Fitur seperti "Jam Aktif" (*Active Hours*) memungkinkan pengguna menentukan kapan *restart* tidak boleh terjadi.
    - Software Update (macOS): Apple mengelola pembaruan macOS secara terpusat melalui fitur "Software Update" di System Settings. Pembaruan ini mencakup perbaikan keamanan dan fitur baru. macOS memiliki reputasi baik dalam hal distribusi pembaruan yang cepat dan konsisten ke perangkat yang didukung.
    - Package Managers (Linux): Sebagian besar distribusi Linux mengelola pembaruan melalui sistem *package manager* (misalnya, APT di Debian/Ubuntu, DNF di Fedora/RHEL, Pacman di Arch Linux). Meskipun sering memerlukan perintah manual (*e.g., sudo apt update && sudo apt upgrade*), banyak lingkungan *desktop* Linux menyediakan alat GUI untuk pembaruan yang mudah. Untuk *server*, alat seperti unattended-upgrades (Ubuntu) dapat mengotomatiskan *patch* keamanan.
    - Over-The-Air (OTA) Updates (Android & iOS): Sistem operasi mobile sangat mengandalkan pembaruan OTA.
        - iOS: Apple memiliki kendali penuh atas *hardware* dan *software*, memungkinkan pembaruan iOS didistribusikan

secara seragam dan cepat ke hampir semua perangkat yang didukung. Pengguna menerima notifikasi dan dapat mengunduh serta menginstal pembaruan langsung dari perangkat.

    - Android: Meskipun Google merilis pembaruan bulanan untuk kernel Android, distribusi ke perangkat pengguna akhir seringkali terhambat oleh produsen perangkat dan operator seluler. Mereka perlu menyesuaikan dan menguji pembaruan untuk perangkat keras spesifik mereka. Inisiatif seperti Project Treble oleh Google bertujuan untuk mengurangi fragmentasi dan mempercepat proses pembaruan.

- Tantangan Pembaruan Otomatis:
    - Kompatibilitas: Pembaruan terkadang dapat menyebabkan masalah kompatibilitas dengan aplikasi atau *driver* lama.
    - Ukuran dan Bandwidth: Pembaruan yang besar dapat memakan banyak *bandwidth* dan ruang penyimpanan.
    - Downtime: Beberapa pembaruan memerlukan *restart*, menyebabkan *downtime* singkat.
    - Kegagalan Pembaruan: Proses pembaruan yang gagal dapat menyebabkan sistem tidak dapat di-*boot*. SO modern sering memiliki mekanisme *rollback* untuk mengatasi hal ini.

C. Manajemen Versi Kernel dan Komponen OS

Manajemen versi adalah praktik fundamental dalam pengembangan perangkat lunak untuk melacak dan mengontrol perubahan pada kode. Dalam konteks sistem operasi, manajemen versi kernel dan komponen OS lainnya sangat krusial untuk stabilitas, keamanan, dan kemampuan *debugging*.

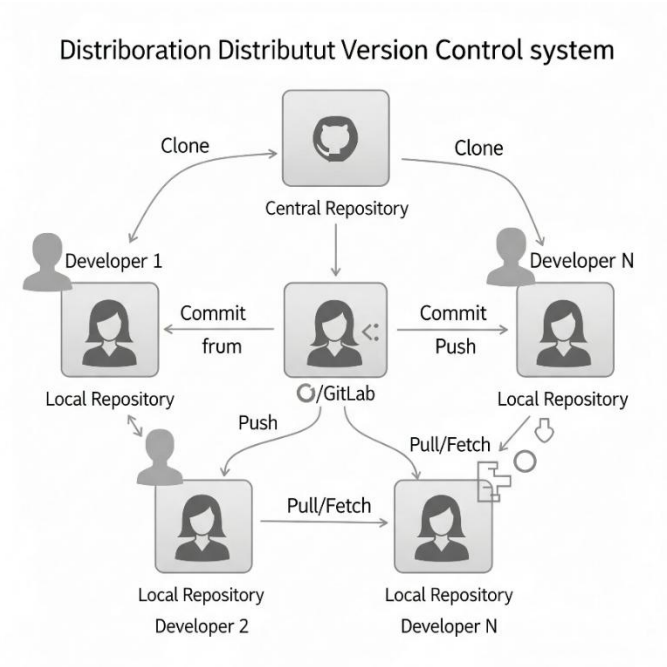
- Pentingnya Manajemen Versi:
  - Pelacakan Perubahan: Mencatat setiap perubahan yang dibuat pada kode sumber, memungkinkan pengembang untuk melihat riwayat perubahan, siapa yang membuat perubahan, dan mengapa.
  - Kolaborasi: Memfasilitasi kerja tim di mana banyak pengembang dapat bekerja pada bagian kode yang sama secara bersamaan tanpa saling menimpa pekerjaan.
  - Kemampuan Rollback: Jika ada *bug* kritis atau masalah performa yang diperkenalkan oleh perubahan baru, sistem dapat dengan cepat dikembalikan (*rollback*) ke versi sebelumnya yang stabil.
  - Debugging dan Analisis: Memungkinkan pengembang untuk mengidentifikasi dengan tepat perubahan mana yang menyebabkan *bug* atau masalah tertentu, mempercepat proses *debugging*.
  - Reproduksibilitas: Memungkinkan pembangunan kembali versi OS atau komponen tertentu di masa lalu, yang penting untuk pengujian dan kepatuhan.
- Sistem Kontrol Versi (Version Control Systems - VCS): Pengembang sistem operasi mengandalkan Sistem Kontrol Versi (VCS) untuk mengelola kode sumber mereka. Alat VCS paling populer saat ini adalah Git.
  - Git: Digunakan secara ekstensif dalam pengembangan kernel Linux, Android Open Source Project (AOSP), dan banyak proyek *open source* lainnya. Git adalah VCS terdistribusi, artinya setiap pengembang memiliki salinan lengkap dari repositori kode, memungkinkan mereka bekerja secara *offline* dan menyatukan perubahan mereka nanti. Ini memungkinkan

model pengembangan yang sangat kolaboratif dan terdistribusi.

- Fitur Git yang Relevan:
  - *Commits:* Merekam perubahan pada kode.
  - *Branches:* Memungkinkan pengembang untuk bekerja pada fitur baru secara terpisah dari *mainline* tanpa mengganggu kode yang stabil.
  - *Merges:* Menggabungkan perubahan dari satu *branch* ke *branch* lain.
  - *Tags:* Menandai rilis stabil atau versi penting dari kode.

- Manajemen Versi Kernel: Kernel sistem operasi (misalnya, kernel Linux) memiliki siklus rilis yang teratur dengan skema penomoran versi yang jelas (misalnya, 5.15.0). Setiap versi mayor membawa fitur baru yang signifikan, sementara versi minor biasanya berfokus pada perbaikan *bug* dan *patch* keamanan. Distribusi Linux, misalnya, memilih versi kernel tertentu dan mengelolanya, memberikan stabilitas dan *patch* keamanan di atas versi tersebut.
- Manajemen Versi Komponen OS: Selain kernel, sistem operasi terdiri dari ribuan komponen lain (pustaka sistem, utilitas, *device driver*, *shell*, aplikasi bawaan). Semua komponen ini juga memiliki versi tersendiri yang harus dikelola dan disinkronkan untuk memastikan kompatibilitas dan stabilitas sistem secara keseluruhan. Dalam ekosistem *open source*, seringkali ada proses yang terstruktur untuk mengelola kontribusi dari berbagai pihak dan memastikan mereka terintegrasi dengan baik ke dalam *release* OS.
- Dampak pada Stabilitas dan Keamanan: Manajemen versi yang buruk dapat menyebabkan masalah serius, seperti *bug* yang sulit

dilacak, ketidakstabilan sistem, atau bahkan celah keamanan yang tidak sengaja diperkenalkan. Oleh karena itu, praktik terbaik dalam manajemen versi, yang difasilitasi oleh alat seperti Git dan metodologi CI/CD, adalah fondasi untuk membangun dan memelihara sistem operasi yang robust dan aman.



(Deskripsi Gambar: Sebuah diagram yang menunjukkan "Repositori Pusat (Central Repository)" (misalnya GitHub/GitLab ikon). Dari Repositori Pusat, tunjukkan beberapa "Pengembang" (Developer 1, Developer 2, Developer N) yang masing-masing memiliki "Repositori Lokal" mereka sendiri. Tunjukkan panah "Clone" dari Repositori Pusat ke Repositori Lokal, panah "Commit" dari Pengembang ke Repositori Lokal, dan panah "Push" dari Repositori Lokal ke Repositori Pusat. Juga, tunjukkan panah "Pull/Fetch" dari Repositori Pusat ke Repositori Lokal. Ini menggambarkan bagaimana pengembang berkolaborasi dan mengelola versi kode.)

BAB 8: MASA DEPAN SISTEM OPERASI

Sistem operasi telah melalui evolusi yang luar biasa, dari sekadar manajer sumber daya menjadi arsitek komputasi yang kompleks dan adaptif. Namun, perjalanan inovasi tidak berhenti di sini. Seiring dengan kemajuan pesat dalam kecerdasan buatan (AI), pembelajaran mesin (ML), proliferasi perangkat *Internet of Things* (IoT), dan kebutuhan akan pemrosesan data di *edge*, sistem operasi terus beradaptasi dan berinovasi untuk memenuhi tuntutan era komputasi berikutnya. Bab ini akan mengeksplorasi tren-tren kunci yang akan membentuk masa depan sistem operasi, membahas integrasi yang semakin mendalam dengan AI/ML, peran krusial SO dalam ekosistem IoT dan *edge computing*, pentingnya model pengembangan *open source*, serta tantangan dan peluang yang menanti di cakrawala pengembangan SO.

A. Integrasi dengan AI dan Machine Learning

Kecerdasan Buatan (AI) dan Pembelajaran Mesin (ML) telah menjadi kekuatan pendorong di berbagai bidang teknologi, dan sistem operasi tidak terkecuali. Integrasi AI/ML ke dalam SO bukan hanya tentang menjalankan aplikasi AI, tetapi juga tentang bagaimana AI/ML dapat meningkatkan fungsionalitas inti dari SO itu sendiri, menjadikannya lebih cerdas, adaptif, dan efisien.

- Optimalisasi Sumber Daya Adaptif: SO modern harus mengelola sumber daya (CPU, memori, I/O, daya) secara efisien untuk ribuan proses dan aplikasi yang berjalan secara bersamaan. AI/ML dapat digunakan untuk:

- Penjadwalan yang Cerdas: Model ML dapat mempelajari pola penggunaan aplikasi dan perilaku pengguna untuk memprediksi kebutuhan sumber daya di masa depan. Misalnya, penjadwal CPU dapat memprioritaskan tugas-tugas yang paling sering digunakan pengguna atau mengalokasikan sumber daya ke aplikasi yang akan segera aktif, mengurangi latensi dan meningkatkan responsivitas.
- Manajemen Daya Prediktif: AI dapat menganalisis data penggunaan baterai dan aktivitas perangkat untuk memprediksi kapan pengguna akan membutuhkan daya dan secara adaptif menyesuaikan mode daya, mengoptimalkan konsumsi energi tanpa mengorbankan pengalaman pengguna.
- Alokasi Memori Dinamis: ML dapat memprediksi pola akses memori dan mengoptimalkan penempatan data dalam *cache* atau memori utama untuk mengurangi *swapping* dan meningkatkan performa.

- Peningkatan Keamanan dan Deteksi Anomali: AI/ML menjadi alat yang sangat ampuh dalam pertahanan siber pada tingkat sistem operasi:
  - Deteksi *Malware* Berbasis Perilaku: Daripada hanya mengandalkan tanda tangan *malware* yang sudah diketahui, model ML dapat memantau perilaku proses, panggilan sistem (*system calls*), dan aktivitas jaringan untuk mengidentifikasi pola-pola yang mencurigakan atau anomali yang mungkin menunjukkan adanya *malware* baru atau *zero-day exploit*.
  - Otentikasi Adaptif: SO dapat menggunakan ML untuk menganalisis pola *login* pengguna (lokasi, waktu, perangkat) dan menyesuaikan tingkat keamanan otentikasi (misalnya,

meminta otentikasi multifaktor jika terdeteksi aktivitas yang tidak biasa).
    - Manajemen Kerentanan Otomatis: AI dapat membantu mengidentifikasi dan memprioritaskan kerentanan dalam sistem, serta merekomendasikan *patch* atau konfigurasi keamanan yang optimal.
- Antarmuka Pengguna yang Lebih Personal dan Intuitif: Integrasi AI juga akan membuat interaksi dengan sistem operasi menjadi lebih alami dan personal:
    - Asisten Virtual Cerdas: Asisten suara seperti Google Assistant, Siri, dan Cortana akan semakin terintegrasi dengan kernel SO, memahami konteks, dan proaktif dalam membantu pengguna. Mereka dapat mengelola tugas-tugas sistem (misalnya, membersihkan *disk*, mengelola notifikasi) berdasarkan preferensi pengguna.
    - Personalisasi Adaptif: SO dapat mempelajari preferensi pengguna (misalnya, aplikasi yang sering digunakan, waktu penggunaan) dan secara otomatis menyesuaikan *layout* UI, rekomendasi aplikasi, atau pengaturan sistem.
    - Interaksi Multimodal: SO akan mendukung kombinasi input suara, sentuhan, *gesture*, dan bahkan pikiran (melalui antarmuka *brain-computer*) yang diinterpretasikan oleh AI.
- Pemanfaatan Hardware AI/ML: Masa depan SO juga akan melibatkan dukungan *native* untuk *hardware* yang dipercepat AI (misalnya, *Neural Processing Units/NPUs*). SO perlu menyediakan API dan *driver* yang efisien untuk memungkinkan aplikasi AI dan fungsi OS yang dipercepat AI untuk memanfaatkan sepenuhnya kemampuan *hardware* ini.

B. Sistem Operasi untuk IoT dan Edge Computing

Proliferasi perangkat *Internet of Things* (IoT) dan kebutuhan akan pemrosesan data di dekat sumbernya (*edge computing*) telah menciptakan segmen pasar baru yang menuntut sistem operasi yang sangat spesifik. SO yang dirancang untuk IoT dan *edge* sangat berbeda dari SO *desktop* atau *server* tradisional karena batasan sumber daya yang ekstrem dan persyaratan fungsionalitas yang unik.

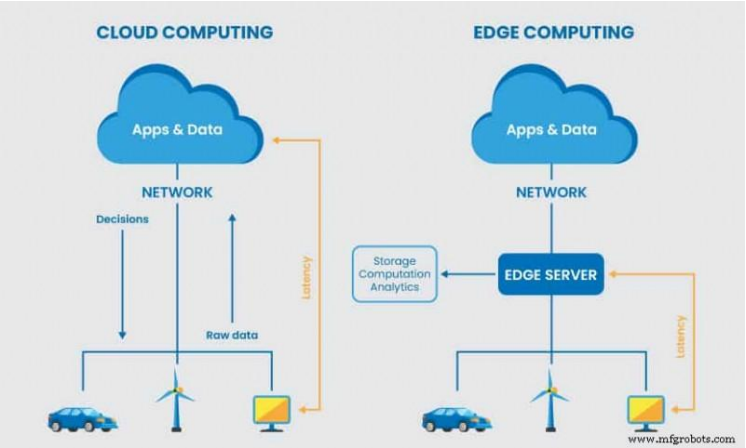
- Kebutuhan Spesifik OS untuk IoT: Perangkat IoT sangat beragam, mulai dari sensor sederhana hingga *smart appliance* yang lebih kompleks. SO untuk IoT harus memenuhi kriteria berikut:
    - Footprint Sangat Kecil: Memori (RAM, *flash storage*) yang terbatas. SO harus efisien dan ringkas.
    - Konsumsi Daya Rendah: Banyak perangkat IoT bertenaga baterai, sehingga SO harus mengelola daya secara ekstrem (misalnya, dengan mode tidur dalam, siklus bangun-tidur yang optimal).
    - Kemampuan Real-Time (Opsional): Untuk beberapa aplikasi IoT (misalnya, kontrol industri, otomotif), SO *real-time*

(RTOS) diperlukan untuk menjamin respons yang dapat diprediksi dalam batasan waktu yang ketat.
    - Keamanan Kuat (Security from the Ground Up): Perangkat IoT seringkali rentan karena sumber daya terbatas dan penyebaran yang luas. SO harus menyertakan fitur keamanan bawaan seperti *Secure Boot*, enkripsi *firmware*, *over-the-air (OTA) updates* yang aman, dan isolasi proses yang efisien.
    - Konektivitas Beragam: Dukungan untuk berbagai protokol nirkabel dan jaringan jarak dekat (Wi-Fi, Bluetooth, Zigbee, LoRaWAN, Thread).
    - Manajemen Jarak Jauh: Kemampuan untuk *provisioning*, *monitoring*, dan *updating* perangkat dari jarak jauh.
- Peran OS dalam Edge Computing: *Edge computing* adalah paradigma komputasi di mana pemrosesan data dilakukan dekat dengan sumber data (di "tepi" jaringan), bukan di pusat data *cloud* yang jauh. Ini penting untuk aplikasi yang membutuhkan latensi sangat rendah, *bandwidth* terbatas, atau privasi data.
    - SO Ringan dan Robust: Perangkat *edge* (misalnya, *gateway* IoT, *mini-server*) memerlukan SO yang cukup kuat untuk melakukan pemrosesan data, analitik, dan bahkan menjalankan model AI/ML, tetapi tetap efisien dan stabil dalam lingkungan yang mungkin tidak terkontrol.
    - Dukungan Kontainerisasi: SO di *edge* sering mendukung *container runtime* (misalnya, Docker) untuk menyebarkan dan mengelola aplikasi secara efisien. Ini memungkinkan pembaruan *software* yang mudah dan isolasi beban kerja.
    - Keamanan Terdistribusi: OS di *edge* harus mampu berpartisipasi dalam model keamanan terdistribusi, menjaga

integritas dan kerahasiaan data yang diproses secara lokal sebelum dikirim ke *cloud*.

- Interoperabilitas Cloud-Edge: OS di *edge* harus dirancang untuk berkomunikasi dan berintegrasi mulus dengan layanan *cloud*, mengirimkan data yang telah diproses atau hanya hasil akhir.

- Contoh OS untuk IoT dan Edge:
  - FreeRTOS, Zephyr OS, RIOT OS: RTOS dan SO *embedded* yang dirancang untuk perangkat *microcontroller* berdaya rendah.
  - Ubuntu Core, Fedora IoT: Versi Linux yang ringan dan dioptimalkan untuk perangkat IoT dan *gateway edge*, sering menggunakan *container* atau *snap packages* untuk manajemen aplikasi.
  - Microsoft Azure Sphere OS: SO berbasis Linux yang sangat aman untuk perangkat IoT, dirancang dengan filosofi "security-first".
  - Google Fuchsia: Seperti yang dibahas sebelumnya, Fuchsia dirancang untuk beradaptasi dari perangkat kecil hingga besar, dengan fokus pada modularitas dan keamanan untuk dunia yang terhubung.



C. Pengembangan OS Open Source

Model pengembangan *open source* telah membuktikan diri sebagai kekuatan pendorong inovasi yang tak terbantahkan dalam pengembangan perangkat lunak, dan ini sangat terlihat pada sistem operasi. Kehadiran dan dominasi Linux, Android, dan berbagai proyek *open source* lainnya menunjukkan bahwa kolaborasi komunitas global adalah masa depan yang cerah untuk SO.

- Definisi dan Filosofi Open Source: *Open source* berarti kode sumber perangkat lunak tersedia untuk umum, memungkinkan siapa pun untuk melihat, memodifikasi, dan mendistribusikannya. Ini didasarkan pada filosofi transparansi, kolaborasi, dan meritokrasi.
- Manfaat Pengembangan OS Open Source:
    - Transparansi dan Auditabilitas: Kode sumber yang terbuka memungkinkan peneliti keamanan dan pengembang untuk meninjau kode secara ekstensif, membantu mengidentifikasi dan memperbaiki *bug* atau celah keamanan lebih cepat daripada sistem *proprietary*. Ini meningkatkan kepercayaan.

- Inovasi dan Fleksibilitas: Model *open source* mendorong inovasi karena siapa pun dapat mengusulkan fitur baru, mengembangkan *patch*, atau membuat *fork* proyek untuk mengejar arah baru. Ini menghasilkan beragam distribusi dan adaptasi yang sesuai untuk berbagai kebutuhan (misalnya, Linux untuk server, desktop, *embedded*, superkomputer).
- Stabilitas dan Keandalan: Ribuan mata yang meninjau kode cenderung menemukan *bug* lebih cepat. Proyek *open source* besar seperti kernel Linux memiliki proses pengujian dan peninjauan yang sangat ketat yang melibatkan kontributor dari seluruh dunia.
- Biaya dan Aksesibilitas: Banyak SO *open source* tersedia secara gratis, mengurangi hambatan masuk bagi individu, startup, dan negara berkembang.
- Kustomisasi: Organisasi atau individu dapat memodifikasi SO *open source* untuk memenuhi kebutuhan spesifik mereka tanpa terikat pada *vendor lock-in*.
- Ketersediaan Sumber Daya Belajar: Kode sumber yang terbuka dan komunitas yang aktif menyediakan sumber daya belajar yang melimpah bagi pengembang dan mahasiswa.

- Tantangan Pengembangan OS Open Source:
  - Fragmentasi (pada beberapa kasus): Terlalu banyak pilihan atau *fork* dapat menyebabkan fragmentasi, meskipun ini juga bisa menjadi kekuatan.
  - Dukungan *Hardware*: Kadang-kadang, *driver hardware* untuk SO *open source* mungkin belum matang atau tidak tersedia untuk *hardware* terbaru dibandingkan dengan SO *proprietary*.
  - Model Bisnis: Menemukan model bisnis yang berkelanjutan untuk mendukung pengembangan *open source* bisa menjadi

tantangan, meskipun banyak perusahaan (Red Hat, Google, Canonical) telah berhasil melakukannya.

- Peran dalam Masa Depan SO: Model *open source* akan terus menjadi kekuatan utama dalam pengembangan SO masa depan, terutama dengan meningkatnya kompleksitas dan kebutuhan akan kustomisasi di lingkungan seperti *cloud*, IoT, dan *edge*. Banyak inovasi penting (seperti *containerization* dan banyak alat AI/ML) dibangun di atas fondasi *open source*.

Linux (Tux si penguin) atau ikon yang merepresentasikan *open source* (misalnya, gembok terbuka dengan kode di dalamnya).

1. Beberapa "Ikon Pengembang" (orang-orang kecil) yang saling terhubung dengan panah atau garis, menuju ke sebuah "Ikon Kode Sumber" di tengah.
2. Di sekitar ikon kode sumber, bisa ditambahkan label seperti "Kolaborasi", "Transparansi", "Inovasi", "Keamanan".
3. Juga bisa menampilkan ikon yang merepresentasikan "Komunitas Global" (misalnya, peta dunia dengan orang-orang di atasnya) yang berinteraksi dengan kode sumber.)*

D. Tantangan dan Peluang ke Depan

Masa depan sistem operasi akan diwarnai oleh tantangan yang semakin kompleks sekaligus peluang inovasi yang belum pernah terjadi sebelumnya.

- Tantangan Utama:
    - Keamanan Siber yang Semakin Canggih: Penyerang terus mengembangkan metode baru. SO harus selangkah lebih maju dengan model keamanan proaktif, *zero-trust architectures*, dan integrasi AI untuk deteksi ancaman. Ancaman terhadap *supply chain* perangkat lunak (misalnya, kerentanan dalam komponen *open source* yang digunakan) juga menjadi perhatian serius.
    - Privasi Data: Dengan semakin banyaknya data sensitif yang dikumpulkan oleh perangkat, SO harus menyediakan kontrol privasi yang lebih kuat, transparan, dan dapat dikelola oleh pengguna, sesuai dengan regulasi global (GDPR, CCPA).
    - Kompleksitas yang Meningkat: Integrasi *hardware* baru (komputasi kuantum, *neuromorphic chips*), *AI/ML*, *IoT*, dan *edge computing* membuat SO semakin kompleks. Mengelola kompleksitas ini tanpa mengorbankan stabilitas dan performa adalah tantangan besar.
    - Fragmentasi Ekosistem: Terutama di segmen *mobile* dan IoT, fragmentasi *hardware* dan *software* terus menjadi masalah yang menghambat pembaruan dan konsistensi.
    - Efisiensi Daya dan Keberlanjutan: Dengan miliaran perangkat, konsumsi daya menjadi isu keberlanjutan. SO harus terus berinovasi dalam manajemen daya yang lebih cerdas untuk mengurangi jejak karbon komputasi.

- Peluang Inovasi:
  - Komputasi Kuantum dan SO Kuantum: Pengembangan SO khusus untuk mengelola *quantum computers* adalah bidang yang baru muncul, menangani tantangan unik dari *qubit* dan algoritma kuantum.
  - Arsitektur CPU Baru: Munculnya arsitektur CPU seperti ARM (di luar *mobile*, juga di *desktop* dan *server*) dan kemungkinan *processor* berbasis RISC-V mendorong SO untuk menjadi lebih *portable* dan adaptif terhadap *hardware* heterogen.
  - Sistem Operasi yang Benar-benar Adaptif/Self-Healing: SO masa depan mungkin akan lebih proaktif dalam mendeteksi dan memperbaiki masalahnya sendiri (misalnya, mengidentifikasi dan mengisolasi komponen yang rusak, memulihkan konfigurasi, atau bahkan memprediksi kegagalan *hardware*).
  - Interaksi Manusia-Komputer yang Revolusioner: Dengan AI, SO dapat memfasilitasi antarmuka yang lebih alami—mulai dari *augmented reality* (AR), *virtual reality* (VR), hingga antarmuka *brain-computer*, mengubah cara kita berinteraksi dengan informasi dan perangkat.
  - SO untuk Komputasi Spasial: Dengan kemajuan AR/VR, kebutuhan akan SO yang dapat mengelola lingkungan komputasi 3D dan interaksi spasial akan meningkat.
  - Keamanan Terverifikasi Formal: Penggunaan metode verifikasi formal untuk membuktikan kebenaran kode kernel dan komponen kritis, meningkatkan jaminan keamanan secara matematis.

Masa depan sistem operasi akan menjadi masa di mana SO tidak lagi hanya sekadar pengelola sumber daya, tetapi menjadi entitas yang semakin cerdas, responsif, dan terintegrasi secara mulus dengan dunia fisik dan digital kita. Kemampuan untuk menyeimbangkan inovasi dengan keamanan, privasi, dan keberlanjutan akan menjadi kunci keberhasilannya..

DAFTAR PUSTAKA

Krutz, R. L., & Vines, R. D. (2010). *Cloud Security: A Comprehensive Guide to Secure Cloud Computing*. Wiley.

Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating System Concepts* (10th ed.). Wiley.

Stallings, W. (2018). *Operating Systems: Internals and Design Principles* (9th ed.). Pearson.

Tanenbaum, A. S., & Bos, H. (2015). *Modern Operating Systems* (4th ed.). Pearson.

Williams, B. (2019). *Computer Systems Architecture: A Networking Approach*. Springer.

Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating system concepts* (10th ed.). Wiley.

Tanenbaum, A. S., & Bos, H. (2014). *Modern operating systems* (4th ed.). Pearson.

Stallings, W. (2017). *Operating systems: Internals and design principles* (9th ed.). Pearson.

Bovet, D. P., & Cesati, M. (2005). *Understanding the Linux kernel* (3rd ed.). O'Reilly Media.

Love, R. (2010). *Linux kernel development* (3rd ed.). Addison-Wesley.

Microsoft. (2023). *Windows 11 documentation*. Retrieved from https://learn.microsoft.com

Apple Inc. (2023). *macOS security overview*. Retrieved from https://developer.apple.com

The Linux Foundation. (2023). *Linux documentation project*. Retrieved from https://www.kernel.org

Google. (2023). *Android open source project (AOSP) documentation*. Retrieved from https://source.android.com

Docker Inc. (2023). *Docker documentation*. Retrieved from https://docs.docker.com

VMware. (2023). *vSphere virtualization guide*. Retrieved from https://www.vmware.com

FreeRTOS. (2023). *FreeRTOS reference manual*. Retrieved from https://www.freertos.org

Open Group. (2023). *UNIX system standards and architecture*. Retrieved from https://pubs.opengroup.org

Intel. (2023). *Intel virtualization technology documentation*. Retrieved from https://www.intel.com

Kubernetes. (2023). *Kubernetes documentation*. Retrieved from https://kubernetes.io