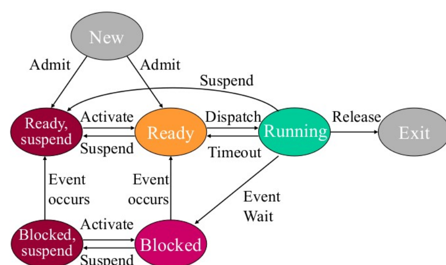
01 – Einführung
Computer teuer und selten -> Ziel optimale Auslastung der Hardware
- Serial Processing. Kein OS, direkte Programmierung der Hardware
  Problem: Fehleranfälligkeit (speziell IO), Lösung Unterprogrammbibliotheken (erste Treiber)
  Problem: Programmstart aufwendig, Lösung Monitor
    Monitor: Erste Art Betriebssystem. Einlesen Job, Ausführung Programm, Fortsetzung
      Enthält dann schon: Treiber, Interrupts (um Programm zu beenden), Sequencer, Interpreter
  Problem: Entweder CPU od IO tätig, rest untätig. Lösung: Multitasking
- Weitere Entwicklung: Billige Hardware -> Rechner pro Benutzer, Fokus auf GUI, Networking..

Betriebssystem “macht Computer (Hardware) verwendbar”. Es ist
- Benutzer / Computer Interface (sitzt auf Hardware auf unterster Software Ebene)
- Virtuelle Maschine (Illusion dass nur ein Programm alleine läuft (unendlicher Speicher, ...))
- Dienstleister (Programmausführung, Filezugriff, Zugriffskontrolle, Fehlererkennung)
- Ressourcenmanager (Keine externe Kontrollinstanz -> selbst für Management verantwortlich)

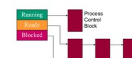
02 – Prozesse und Threads
Ziel: “Gleichzeitiges” Ausführen von Programmen auf einem Rechner
Programm: Statische Beschreibung, der Code
Prozess: Ein Program in Ausführung, zusätzlich Verwaltungsdaten: Variablen, Stack, Puffer
    Kontext: Aktueller Zustand (Prog count, Register); Daten zur Prozessverwaltung
Trace: Instruktionen die für einen Prozess ausgeführt werden, geht auch für mehrere
  -> Ausgeführte Prozessoren überlappen, immer stückweise ausgeführt. Dzw Dispatcher Code

Erzeugung Prozesse: BS baut Datenstruktur auf und allokiert Speicher. Wann? Login eines Benutzers, BS: Ausführung Service, Erzeugung durch anderen Prozess (child)
Beenden Prozess: Logout Benutzer, Fehler in Abarbeitung, Halt Instruktion, BS Service request



Running: Prozess wird ausgeführt
Ready: Prozess bereit zur Ausführung (wartet auf Zuteilung)
Blocked/Waiting: Prozess nicht laufbereit, wartet auf Ereignis
New: BS hat Prozess erstellt, jedoch nicht laufbereit (zb Vermeidung Überlastung durch zu viele Prozesse)
Exit: Erreicht durch Terminierung, Prozessinformationen werden gelöscht.

Queuing Modell: Admit -> Ready Queue. Dispatcher nimmt aus Ready Q und dispatched an CPU.
Durch Timeout nächster, oder durch Block. Je blocking Event eine eigene Q (Blocking/Waiting Q).
Queues oftmals verkettete Liste über Process Control Block (Querverweis)



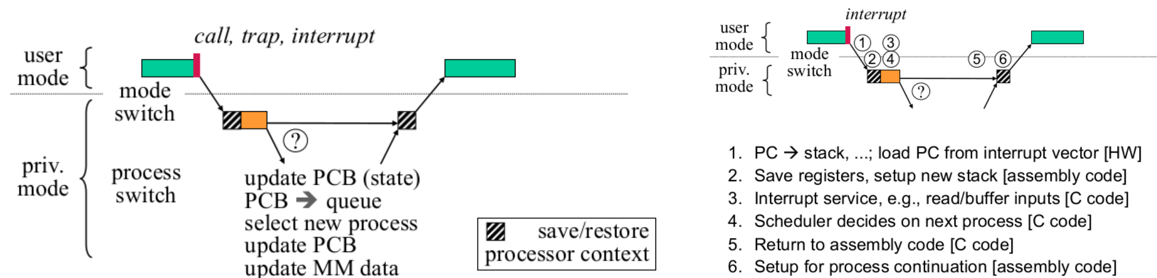
Process Switch: Wenn BS in CPU Besitz, Wenn supervisor call (IO), Trap (Fehler), Interrupt
Swapping: Auslagerung Prozessdaten auf Sekundärspeicher (suspend Zustände), Gründe: Performance, Debugging, Auslagerung Hintergrundprozesse etc.

Kontrollstrukturen Betriebssystem: Verwaltet Tabellen für Prozesse und Ressourcen:
  Memory Tables, IO Tables (Buffer etc); File Tables, Process Tables (Verweis auf Process Image)

Process Image: Im Virtual Memory (kann tlw ausgelagert sein) muss nicht zusammenhängen
- Process Control Block (PCB): Process Identification: Id, Benutzerkennung (für Rechte), PID
    Processor State Info: Register, PC, Stack Pointers
    Process Control Info: Zustand, Priorität, Ereignis worauf gewartet; Querverweis auf andere Prozesse; verwendete IO, Privileges, Interprozesskommunikationsdaten

- System Stack: Parameter und Call Address von System Calls (getrennt von User Data! Schutz)
- User Data: Modifizierbarer Bereich: Daten, User Stack,
- User Program
- shared address space

Execution Modes: Privileged (System Mode), User Mode;MSwitch nicht automatisch prozesswitch
Um Datenstrukturen von BS zu schützen (zB PCB von Process Image).



Verschiedene Möglichkeiten BS Code und Prozesse zu strukturieren:
- Nonprocess Kernel: Nur Userprogramme als Prozess, BS Code getrennt, immer verlassen Context
- BS Ausführung in User Prozessen: BS ist Sammlung von Routinen die von Usercode aufgerufen werden kann; Fast alle BS Routinen im Prozesskontext; Verlassen Kontext nur bei Prozess Switch Process Image: BS Code u. Daten im Shared Address Space, eigener Kernel Stack
- Prozessbasiertes BS (Auch Microkernel): BS Sammlung von Systemprozessen. Nur Basisservices (prozess switch, IO, Interrupts, basic memory management) nicht als Prozess. Pro: Klein, Flexibel

Kernel Architekturen: Monolithic (Ein großer, unübersichtlicher Kernel); Layered (Unix Schalenmodell, auch nicht optimal) -> Heute oft modularisiert

Threads: Entkoppelung Process (Ressourcenverwaltung), Thread: (Einheit für Dispatching)
Thread: Ausführungszustand (Running, Ready, Blocked, kein suspended); Kontext, Stack, Variablen
→ Process: PCB, User address space; Thread: TCB, User Stack, Kernel Stack
User Level Threads (komplett im User Space); Kernel-Level Threads (Kernel Level Thread API)

03 – Mutual Exclusion und Synchronisation
Paralleler Zugriff auf gemeinsame Daten -> disziplinierter Zugriff, geordnete Abarbeitung, sonst Fehlverhalten (Inkonsistenz, nicht eindeutige Ereignisfolge)
- Wechselseitiger Auschluss (Mutex): Verhinderung gleichz. Ressourcenzugriff; Atomizität
- Bedingungssynchronisation (Condition synch): Erreichen definierte Abfolge von Operationen
  4 mögl Kombis: unabhäng; Abfolge (C); Konsistenz (M); Abfolge + Konsistenz (C + M)

BS Aufgaben: Mutex; Datenkonsistenz; Regelung Abfolge Ressourcenvergabe; Verhinderung Deadlock und Starvation
kA: Eintritt: Prolog; Austritt: Epilog. Kein Prozess gleichzeitig in kA (-> Mutex)
Anforderung Kritischer Abschnitt Lösungen: Mutex; Progress (Entscheidung über nächsten Prozess nicht unendlich lange verzögert (Lifelock)); Bounded Waiting (Nach Req für kA Anz Eintritte in diesen für andere Prozesse beschränkt -> keine Starvation)

Arten Lösungen: Softwarelösungen (viele Annahmen); Hardwareunterstütze Lösungen; Höhere Synchronisationskonstrukte (Semaphore, Monitore, Message Passing).

Busy Waiting: Warten verschwendet CPU Zeit. Besser -> Wartende in Blocked Queues.

<u>Mutual Exclusion</u>:

-> *Dekker Algorithmus (bs03)*
-> *Peterson Algorithmus (bs03)*
-> *Bakery Algorithmus (bs03)* – (für mehr als 2 Prozesse möglich, aber immer noch beschränkt)
(number[i], i) < (number[j], j) symbolisiert dass wenn number[i] == number[j] mit indices verglichen wird.

Hardwareunterstützte Lösungen: Interrupt disable während kA. -> Einschränkung des BS.
-> Mächtigere atomare Maschineninstruktionen: test and set; exchange (swap)

Semaphor: Abstrahiert Busy-Waiting. Besteht aus int value, prozess queue und flag. Operationen:
- init: setzt value auf Wert >= 0. (Mutex auf 1; Auch andere Werte möglich)
- wait: Dekrementiert value um 1. Wenn value nun < 0 dann wird Prozess in die queue gesetzt.
- signal: Incrementiert value um 1. Wenn value nun <= 0, dann gibt es mind. 1 prozess in queue → wird nun aus queue in ready queue verschoben (wieder dispatchable)

Vom BS unterstützt

Prozess aus Queue auswählen: zB mit FIFO -> fair, erfüllt bounded waiting (keine starvation)
Intern busy waiting mit flag für kA (testset), aber Overhead vernachlässigbar (code kurz).

Binary Semaphore: Nur Werte 0/1; Counting Semaphore: beliebige, oft nur positive, Werte

Bedingungssynchronisation:
- Einmal P1 vor P2: init(S, 0), P1: signal(S) P2: wait(S)
- Abwechselnd: init(S1, 1), (S2, 0); P1: w(S1), write, sig(S2); P2: w(S2), read, sig(S1)
  -> Auch gleichzeitig Mutex auf Daten

Producer-Consumer Problem: Schreiben/Lesen aus Buffer: -> Konsumenten können nur lesen wenn Daten da; außerdem Mutex beim Buffer manipulieren. Prod: w(M), write, s(M), s(C). Cons: w(C), w(M), read, s(M).
Nichtendlicher Buffer -> als Ringbuffer.
Achtung: Reihenfolge waits wichtig. Generell Mutex eher möglichst nah an kA, Bedinungssynch eher weiter außen, sonst Deadlock möglich. Reihenfolge von signals egal

Reader-Writer Problem: Beliebige Leser parallel, Schreiber exklusiven Zugriff
- (1) readcount (rc) -> erster leser blockiert mutex. Epilog: letzter löst mutex; Priorität: Reader
- (2) writecount(wc) und reader mutex. Leser und Schreiber warten auf reader mutex. Priorität: schreiber. Damit Leserqueue nicht so lang ist (für nächsten schreiber): zusätzlicher mutex drumherum bei Leser -> Leser sammeln sich da, eigtl. Queue enthält nur 1. Leser.
- (3) vermeidung writecount: Vermeidung starvation der Leser -> *Folien(bs03) !*

Dining Philosophers: Triviale Lösung links, dann rechts -> deadlock
- Zusätzlicher Semaphore mit init(n-1) (erlaubt aufnehmen von nur n-1 Gabeln)
- Mindestens 1 Prozess nimmt Gabeln in umgekehrter Reihenfolge auf
- Atomare wait Operation für mehrere Semaphore

Nachteile Semaphore: Operationen über Prozesse verteilt -> unübersichtlich, leicht Fehler passieren
-> Monitore -> (*bsp bs03*)
“kapselt” Shared Memory, Prozeduren, Initialisierung und Synchronisation
- Zugriff auf lokale Variablen, shm über Monitor Prozeduren (sind “atomar”)
- Mutex: von Monitor erfüllt (max 1. Prozess zu jedem Zeitpunkt im Monitor)
- Bedingungsynchronisation: Wartebedingungen mit Bedingungsvariablen (Zugriff nur lokal im Monitor - cwait, csignal (wenn kein Prozess wartet, keine Aktion =/= Semaphore))

Message Passing: Methode für Interprozesskommunikation (IPC). (*bsp bs03*)
Nachricht ist atomare Datenstruktur (Immer ganze Struktur oder gar nichts), Charakteristika:
- Synchronisation: send/receive: blocking / non-blocking
- Datenverwaltung: Warteschlange / Empfangene Daten überschreiben alte Werte
- Addressierung: 1:1 vs 1:N; direkt vs indirekt (Portaddressierung vs Mailbox)

Locks: ähnlich Semaphore, enter(lock) (wait); release(lock) (signal)

Sequencer und Eventcounts: -> (bsp *bs03*) Ermöglicht zyklische abwechselnde Prozesse, etc.
- Eventcount: Ereigniszähler: advance(E) E+1, init: 0; await(E, value) blockiert bis E >= val
- Sequencer: “Nummernserver”, gewährt Nummer und inkrementiert danach, init: 0

04 – Deadlock

Permanentes Blockieren Prozesse die um Ressource konkurrieren / Zyklischer Ressourcenkonflikt
- erneuerbare Ressourcen: zB Speicher, kann wieder zurückgesetzt werden.
- konsumierbare Ressourcen: sind dann “weg”, zB Nachricht aus Mailbox

Keine universelle Lösung; Keine automatische Prüfung auf Deadlocks möglich (Halteproblem!)
Gibts auch in “echt”: bsp Kreisverkehr mit Rechtsvorrang. Verhinderung: Nachrang Einfahrende

Prozessfortschrittsdiagramm Visualisierung. Ungültige Abarbeitungsbereiche markiert. Pfeil kann nur nach oben / rechts (keine Rückwärtsabarbeitung möglich); Wenn in einen Bereich wo rauskommen nicht mehr möglich ist (Folie 07) -> Deadlock.

4 Deadlock Bedingungen: 3 Vorraussetzungen, Auftreten einer bestimmten Ereignisfolge (4)
(1) Mutual Exclusion
(2) Hold and Wait (Prozess kann Ressourcen halten, während er auf andere wartet)
(3) No Preemption (zugewiesene Ressourcen werden Prozessen nicht weggenommen)
(4) Circular Wait: Kette Prozesse, jeder hält mind. 1 Ressource die von anderen benötigt
-> Deadlock genau dann wenn Circular Wait nicht auflösbar; Nicht auflösbar wenn 1-3 gilt

Behandlung von Deadlocks:
Deadlock Prevention:
- Indirect: Verhindern Bedingungen 1-3: Mutual Exclusion: Ist eine Zielsetzung -> geht nicht
  Hold and Wait: Prozesse fordern alle Ressourcen auf einmal an (ganz oder gar nicht)
    Cons: Verzögerung möglich, Schlechte Nutzung allokierter Ressourcen
  No Preemption: a) Prozess gibt Ressourcen frei wenn er eine weitere nicht bekommt
    b) Anforderungen können anderen Prozess Ressourcen entziehen (problematisch)
- Direct: Verhindern Circular Wait. Lineare Ordnung Ressourcenarten O(drive) < O(disk) < …
  Alle Ressourcen einer Ordnung müssen in einen Schritt angefordert werden. Anschließend nur mehr Anforderungen mit größerer Ordnung zugelassen; Con: Ineffizient: Zurückweisung Anf.en

Deadlock Avoidance: Bedingungen 1-3 erlaubt. Selektive Vergabe von Ressourcen:
- Process Initiation Denial: Nicht gestartet wenn Anforderung zu Deadlock führen könnte
  Prozess nur gestartet wenn Anz R i Kategorie >= Anz R allokiert + Anz R die Prozess haben will
  Con: Annahme alle Prozesse wollen alle Res gleichzeitig -> Einschränkung Parallelität
- Ressource Allocation Denial: Anforderung verwehrt wenn zu Deadlock führen könnte
  Banker’s Algorithm (Folien *bs04*). Safe State -> Kein Deadlock möglich; Unsafe State -> Deadlock möglich, muss aber nicht (defensive Annahmen)

Nehmen Unabhängigkeit Prozesse an; Kein Deadlock Schutz bei Bedingungssynchronisation

Deadlock Detection: Ressourcenanforderungen immer gewährt sofern Ress vorhanden
- BS Vorkehrungen: Algorithmus zum erkennen + Strategie zur Behebung (Recovery)

-> Bsp Algorithmus in Folien (*bs04*)
-> optimistische Annahme Prozesse keine weiteren Ressourcen zur Fertigstellung
-> Überprüfung bei jeder Ressourcenanforderung -> CPU intensiv
Deadlock Recovery:
- Abbrechen aller beteiligten Prozesse (häufig)
- Rollback beteiligter Prozesse zu vorigen Zeitpunkt (Deadlock kann sich wiederholen)
- Abrechnen einzelner bis Deadlock weg (Aufrufen Detection Algo)
- Ressourcen anderen Prozessen schrittweise entzogen und an andere vergeben bis kein Deadlock

Integrierte Deadlock Strategie: Kombinieren der Ansätze. Gruppieren Ressourcen in Klassen und ordnen. Circular Wait Prevention zw Klassen. Innerhalb Klassen geeignete Vermeidungsstrategie

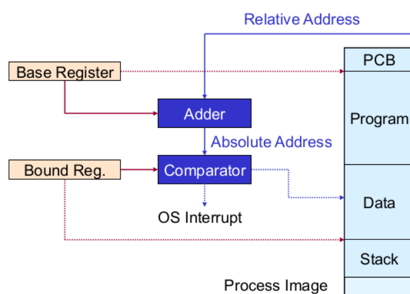
05 – Memory Management

Anforderungen: Partitioning (Speicheraufteilung auf Prozesse); Relocation; Protection; Sharing (Gemeinsamer Zugriff auf Speicher); Performance

Partitioning:
- Fixed Partitioning: Fixe Unterteilung. Auslagern von Prozess wenn alles belegt. Bei Programmen größer als Partition -> Programmierung von Overlays. Potentiell verschw. von Speicher in einer Partition (interne Fragmentierung)
- Dynamic Partitioning: Zw Partitionen entstehen Löcher (Externe Fragmentierung) -> Compaction (Verschieben um wieder Platz haben) Placement Strategies beim Einfügen: Best / First / Next fit
- Buddy System: Kompromiss: Unterteilung in Partitionen Größe 2^k. Mit min <= k <= max val. Wenn 128m in 1G -> 1g in 2*512, 512 in 2*256, 256 in 2*128, einer der 128 wird belegt. Wenn wieder frei -> zusammenfassen zu nächstgrößeren Block

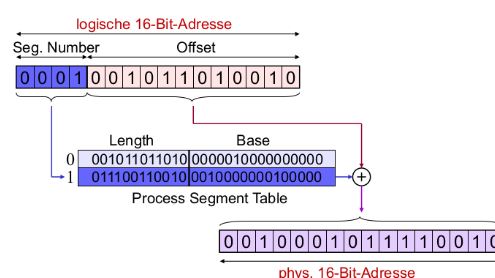
Relocation: Referenzen auf physikalischen Speicher müssen veränderbar sein.
- physische Adresse: Absolute Position im Hauptspeicher
- logische Adresse: Referenz auf Position im Speicher, unabhängig von Speicherorganisation
- relative Adresse: Position relativ zu Punkt im Programm (= logische Adresse)

Einfache Adressübersetzung (HW):



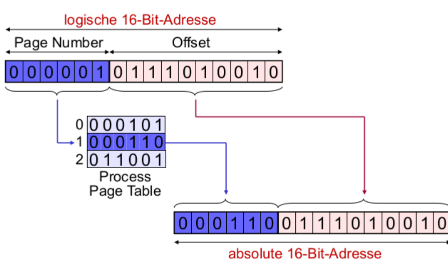
Startadresse Prozess in Base
Speicherschutz: Über Bound

Segmentierung:

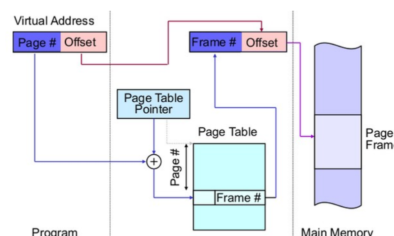


Unterteilung Programm in Blöcke untersch Länge
keine interne aber externe Fragmentierung
Sichtbar für Programmierer

Paging: Unterteilung Hauptspeicher in Frames gleicher Größe; Prozesse in Pages gleicher Größe

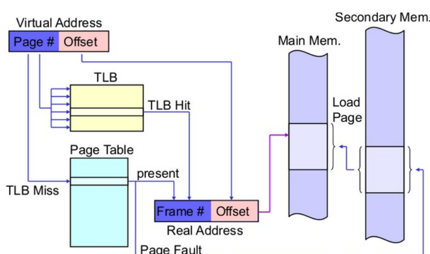


Pagetable / Prozess für jede Page die aktuelle Framenummer
Free Frame List verweist auf freie Frames im Speicher



Virtual Memory:
Aufspaltung Prozesse in Seiten die nicht hintereinander im Speicher sein müssen. Nicht alle Seiten müssen sich im Hauptspeicher befinden, nur aktuell benötigte (Resident Set) -> Nutzt Lokalität: Normalerweise arbeitet Programm oft im lokal benachbarten Speicher
- Page Fault: Adresse nicht im Hauptspeicher wird referenziert
- Thrashing: Mehr Zeit mit Laden von Seiten als wie Abarbeiten von Aufgaben (Häufige PageFault)
→ Suspended Process: Etwa wenn zu wenig Seiten im Resident Set vorhanden
-> Einträge in Pagetable nun: PresentBit (Seite in Hauptspeicher?); Framenumber; Modified Bit (Seite seit Laden verändert); Control Bits (Locking, Kernel/User Page)

Page Table kann je Prozess sehr groß werden, Optimierungen:
- Multilevel Page Table: Teilung 1. Adress Teil: Root Page table -> Page of Page table p1|p2|Offset
- Inverted Page Table: Table in RAM wo (pid,page) für jedes Frame im RAM gespeichert (Index)
  logische Adresse nun pid|p|offset. 2 Zugriff je Addr nun nötig -> Translation Lookaside Buffer
  klein Cache in CPU zuletzt verw Seiten. Funktioniert da Lokalität. Löschen bei Context switch



Gesamt mit Virtual Memory, Paging und TLB

Wann Seite laden? -> Fetch Policy
Demand Paging: Wenn Adresse der Seite referenziert (Anfang viele Page Faults)
Prepaging: Lädt im Voraus, durch Lokalität motiviert

Replacement Policy: Wann Seite bei Laden neuer Seite ersetzt wird.
- OPT Policy: Optimal, schaut in die Zukunft, nicht implementierbar, nur als Referenz
- LRU Policy: Least recently used. Nah an OPT. Implementierung aufwändig (Speichern, Suche)
- FIFO Policy: Einfach (zyklisch schreitender Pointer); Con: Oft verwendete Seite kann älteste sein
- Clock Policy: Ring, zeiger zyklisch durch, wenn 1 → auf 0, weiter; wenn 0 -> ersetzen. Fast LRU

Working Set Strategie: Beobachtung Frames für Prozess / Zeitfenster -> Anpassung. Aufwendig, Realität: Beobachtung PageFaults / Zeiteinheit. Wenn über Grenzwert -> suspend Prozess

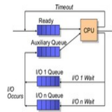
Protection: Durch Paging: PageFault der nicht auf ausgelagerte Seite zeigt. Wenn Zugriff erwünscht: Protection Keys. (evtl Nachlesen Varianten in Folien *bs05*)

06 – Scheduling
Bestimmt Abarbeitungsfolge Prozesse. Versch. Optimierungsziele: Durchsatz, Auslasung, Fairness..
Ebenen:
- Long-term: Bei Erstellung von Prozessen, Mix CPU, I/O intensiver Prozesse
- Medium-term: Ein / Auslagerung von Prozessen
- Short-term: Bestimmt nächsten Prozess zur Ausführung

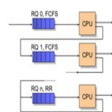
<u>Short-term Scheduling</u>:
CPU Scheduler oder Dispatcher. Aktivierung wenn Switch angebracht sein kann, Kriterien: Systemcalls / Traps; I/O Interrupt / Signale; Timeout-Interrupt

Scheduling Kriterien: Performance: Useroriented: Response Time, Systemoriented: CPU Utilization
Other: Useroriented: Predictability; Systemoriented: Fairness…

Schedulingstrategien:
- Selection function: Bisherige Verweildauer, bisherige Ausführzeit, Fertigstellungszeitpunkt
- Decision mode: Non-preemptive: Prozess läuft durch; Preemptive: Prozess gestückelt



First Come First Served: Am längsten in Queue wartet, non-preemptive. Begünstigt lange Prozesse,
Round Robin: Am längsten in Queue, preemptive mit Timeout. Timeoutlänge: Länger als Clock
Interrupt + Scheduling; Benachteiligt I/O intensive Prozesse
Virtual Round Robin: Zusätzlich priorisierte Auxiliary Queue mit Prozessen nach I/O Block
Shortest Process Next: Nächster: kürzester; non-preemptive. Bessere Response time als FCFS
Con: Schätzung Abarbeitungszeiten; Starvation langer Prozesse
Shortest Remaining Time: preemptive, nicht so viele Interrupts wie RRobin. Starvation möglich



Highest Response Ratio next: größte RR: (bish warte+s)/s s..geschätzte Service time (nötig = Con).
Feedback Scheduling: Je neuer desto mehr Prio (Queue je Prio); preemptive. Starvation möglich

Real-time Scheduling: Hard Deadlines - kritische Systeme, oft periodische (wiederkehre) Prozesse
meist preemptive Scheduling; Schedulability Test (Überprüfung alle rechtzeitig abgearbeitet können
Earliest Deadline First: mit frühester Deadline zuerst; preemptive; minimiert deadline misses

07 – Ein/Ausgabe und Disk Scheduling
Ein/Ausgabe: Vielzahl Gerätearten. 3 grundsätzliche Arten: Mensch Interface (Anzeige, Tastatur, etc); Maschinen Interface (Laufwerk, Sensoren, ..); Kommunikation zB Netzwerk

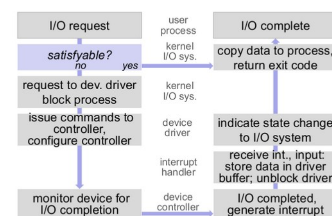
Merkmale I/O Geräte: Datenrate; Anwendung; Ansteuerung (Poll / Interrupt); Transfer-Einheiten (Zeichen / Blöcke); …
I/O Funktionen: Programmed I/O: Prozess schickt IO Kommando und wartet mit Busy Waiting
Interrupt-driven I/O: Prozess -> Kommando; BS arbeitet weiter; IO Modul unterbricht wenn fertig
Direct Memory Access (DMA): Prozess -> Kommando an IO Modul; Modul kopiert autonom Daten zu Speicher, unterbricht CPU wenn fertig. Effizient: Kein Kontextwechsel nötig
I/O Channel: Erweiterung zu DMA. IO Instruktionen am RAM, Abarbeitung durch IO-CPU.

OS Design: Kompromiss zwischen: Effizienz (gerätespezifische Lösungen); Flexibilität (einheitliche Schnittstellen, bsp File Ops für Programmierer)
-> Schichtung: Logical IO (abstrakt) -> Device IO (IO Kommandos) -> Scheduling und control

IO bei Geräte mit File System: Logical IO aus 3 Komponenten: Directory Management (Directory Ops; Dateiname -> Dateireferenz); File System (Dateioperationen, open, read, ..); Physical Organisation (Übersetzung in Spuren / Sektoren)

Blocking (Prozess von Running in Blocked Queue) vs Non-block: (sofort, return zB Anz Bytes)
Synchron (Aufrufkette, prozess wartet) vs Asynchron (Prozess wartet nicht, dann Interrupt)

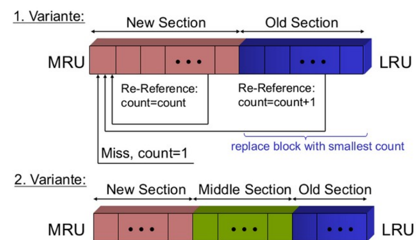
Synchroner IO Request, beispielhaft:



Puffern von IO Anfragen: Zwischenspeicher Daten bei IO Transfer
-> Reduktion Overhead, Entkopplung Prozess zu IO, Abfangen Spitzenlast (wenn Programm Burstschreibverhalten etwa)
Arten: Einfach; Doppelt; Ring;

Disk Scheduling: Seek Time (Zeit Arm zu Spur); Rotational Delay (Zeitverzögerung bis Anfang Sektor); Transfer Time (Zeit Übertragung Daten). -> Average Access Time
Strategien: Ordnung Requests zur Minimierung seek time:
- Priority (kann viel bew verursachen); FIFO,
- shortest service time first: am wenigsten Bewegung vom Kopf
- scan: bewegt sich von außen nach innen, nimmt requests am weg mit; Häufing Randrequests
- c-scan: requests nur in einer richtung, andere leerfahrt. -> Bessere Verteilung
- n-step-scan: zusätzlich vor gruppierung der requests die nun bearbeitet werden sollen -> fairer

Disk Cache: Frequency-Based Replacement:



08 – File Management

Motivation: Prozess nur eingeschr Infomenge in Arbeitsraum
Arbeitsraum nur während Lebensdauer Prozess verfügbar -> Files
speichern große Infomengen; Persistenz; Zugriff durch mehrere Prozesse
Benutzersicht: Dateiname, Directorystruktur, Dateiaufbau (Text, Records,..) (logische Sicht)
vs BS Sicht: Physcial Blocks; IO; … (wie intern effizient gestalten)

Dateiorganisation: (Heute Files meist streams; gibt aber BS die Struktur unterstützten)
Ziele: Zugriffszeit kurz; Aktualisierbarkeit; Geringer Platzverbrauch; Zuverlässigkeit
- Pile: Records variabler Länge; Chronologisch; schwer zu durchsuchen
- Sequential File: Records fixer Länge; Key Feld
- Indexed sequential File: Ein Index; Temp Overflow Datei für neue Datensätze
- Indexed File: Mehrere Indices (je Attribut) zur besseren Durchsuchbarkeit
- Hashed File: Hash Funktion bestimmt Position; Mit Overflow Datei für neue Datensätze

File Types: Regular (Ascii; Binär (Executables vom BS als solche erkannt)); Directories; Character Special Files (Repräsentation IO Geräte); Block Special Files (Repräs. Platten)
File Attributes: Creator, Owner, Protection, Flags (Read-only, hidden, Archive, ..), Creation/Modified time, Current size
File Names: Anz Zeichen; Groß/Kleinschreibung; File Name Extensions
File Operationen: Create, Delete, Open, Close, Read, Write, Seek, …

Dateien in Verzeichnis. Baumstruktur. Verzeichnis ops: Create,Delete, .., link/unlink, zugriffsrechte

Implementierung:
Disk Unterteilung: (MBR | Partitions ..) Partition: Boot Block | Free mgmt |..| Dirs
MBR: Boot Code, Partition Table (start/end/active); Partition: Unabhängige Filesysteme
Bei Start: BIOS exec MBR Code -> aktive Partition -> Boot Block von Partition: Laded BS

Datei Implementierung: Besteht aus Sammlung Blöcke
- Contiguous Allocation: Blöcke Hintereinander, Lesen gut, Vergrößern problematisch, CD/DVD
- Chained Allocation: Je Block: Zeiger Nachfolger; Con: Verschw. Platz durch Zeiger
- Indexed Allocation: Pointer in File Allocation Table (FAT); Con: Platz für FAT im RAM
- Index Nodes: Pointer in Datenstruktur je File (zusammen mit Attributen). Nur im RAM wenn gebraucht. Kann auf Block zeigen, der wieder aus Pointer besteht usw. (für große Dateien)

Blöcke in Sequential Files:
- Fixed Blocking: Blöcke mit Records auffüllen, zu kleiner Rest bleibt ungenutzt (Verschnitt)
- Variable-length spanning Blocking: Records variabler Länge, können über mehrere Blöcke gehn
- Variable-length unspanned Blocking: Nicht über Blöcke, wieder Verschnitt

Directory Implementierung: Lokalisieren Root Dir.
-> Fixe Position am Partitionsanfang / Unix(erster I-Node verweist auf Rootdir) / Win(Boot Sector -> Master File Table)
File-Attribute: Im Directoryentry (Windows) / Im I-Node (Directoryentry: (name, inode nummer))

File Block Size: Speichernutzung vs. Zugriffszeit. Auf Unix durch Untersuchungen: 2 KB
Verwaltung freie Blöcke: Chained Free Portions: Mit Zieger verb; mit der Zeit Fragmentierung
- Bit Tables: Bitvektor mit Bit je Plattenblock. Geringer Platzbedarf, guter Überblick
- Indexing: Freie Blöcke als Datei betrachtet. Effizient für alle Dateiverfahren

Performance: Disk caching: Ausnutzung Lokalität;
- Block Read Ahead; Optimierung Kopfbewegungen – bsp. Verteilung I-Nodes statt hintereinander

09 – Security
Strateg, Vorkehrung, Tools um Vertraulichk, Integrität, Verfügbarkeit von Infos Org gewährleisten
Security Objectives (CIA-Triad):
- Confidentality, Integrity, Availability. Zusätzlich Authenticity, Accountability
Security Threat Types: Passive (Abhören, Confidentality); Active (Aktive Manipulation, DoS)
Security Threads: Interruption, Interception, Modification, Fabrication
Intrusion: Ziel: Verschaffen Zugang zu System / Erhöhung Privilegien; Wege: Lücke, Passwspionag
Intruders: Nicht versierte, Technische Insider, Gezielte Versuche Bereicherung, Industriespionage

Malware:
- Virus: In Programm versteckter Code, kopiert sich in andere Programme
- Wurm: Verbreitet sich selbst über Netz etc
- Trojaner: Programm mit Funktionalität dass auch bösartigen Code enthält
- Logic Bomb: Wird erst nach Bedingung aktiv, bsp. Nichteinloggen für eine Zeit (gefeuerte MA)
- Trapdoor: Geheimer Einstiegspunkt
- Port Scan;   DoS; DdoS

Typische Methoden von Attacken: Bsp -> Stack/Buffer Overflow (*bs09*)
- Anfordern, Auslesen von Speicher
- Aufruf unerlaubter Syscalls bzw Aufruf mit sinnlosen Parametern
- Verwendung Abbruchtasten während Login
- Modifikation BS Strukturen im User Space
- Social Engineering

Design Principles Security: Open Design, Sane-Defaults, Least-Privlege, Einfachheit, Acceptance

User Authentication: Besitz (chipkarte), Attribut (Fingerabdruck), Wissen (Passwort):
Maßnahmen gegen Attacken: OTP, Zeitablauf, Schutz zu viele Versuche, Anzeige letzter Login

Protection: Sicherstellen des kontrollierten Zugangs zu Programmen / Daten auf Computersystem
Protection Domains: Gruppierung; Unix: Durch UID/GID von Prozess
Access Matrix (Rows: Domains). Implementierung ACLs / Capability Lists (mit Tickets)
Mechanism von BS (Enforce Policy) vs Policy (Benutzer bestimmt wie Obj verwendet)

Lock-Key System: Jedes Objekt hat Locks (eindeutiges Bitmuster), jede Domain Keys.

Bell and LaPadulas Modell: Sicherheitsklassifizierung (SC). Top secret, secret, public, ..
Read: Nur SC(S) >= SC(O); Write: SC(S) <= SC(O) (nicht nach außen leaken) - *property.
-> read-write nur bei SC(S) = SC(O)

Geschlossenes System: ok, Offenes System: Kryptographie
Architecture: Subject -> Reference Monitor (checks access rights, audits) -> Object