# Pengenalan Sistem Operasi

Pengantar   
Sistem Operasi

Dosen : Arfiani Nur Khusna, S.T., M.Kom

Email : arfiani.khusna@tif.uad.ac.id

Referensi

Hariyanto, B. (1997) Sistem Operasi. Penerbit Informatika Bandung.

Silberchatz, A., Galvin, P. B. (1999) Operating System: Design & Implementation.

Russinovich, M.E., Solomon, D., (2004), Microsoft Windows Internals, Microsoft Press, America.

De el el

Proses Pembelajaran

Anda Sepakat?

UAS=30

UTS=25

UK, Tugas=20

Praktikum=25

Tot=100%

Apa itu Sistem Operasi?

Perangkat lunak yang bertindak sebagai perantara antara pemakai komputer dan perangkat keras

Sasaran Sistem Operasi

Menjalankan program-program dari user dan membantu user dalam menggunakan komputer

Menyediakan sarana sehingga pemakaian komputer menjadi mudah (convenient)

Memanfaatkan perangkat keras komputer yang terbatas secara efisien (resource manager)

Abstraksi Komponen Sistem

Komponen Sistem Komputer

1. Hardware – menyediakan “basic computing resources” (CPU, memory, I/O devices).

2. Operating system – mengendalikan/mengkoordinasikan penggunaan hardware diantara berbagai aplikasi/program dari user.

3. Applications programs – menggunakan sistem resource yang digunakan untuk menyelesaikan masalah komputasi dari user (compilers, database systems, video games, business programs).

4. Users (people, machines, other computers).

Sistem Operasi

Resource allocator

mengatur resource

mengalokasikan dan mengontrol pemakaian resources dari berbagai program/aplikasi.

Control program

Mengendalikan eksekusi user program dan pemakaian sistem resource (contoh : operasi pada I/O device) => handal, reliable, terlindung.

Kernel

Sistem program yang berjalan (“ada) terus menerus selama komputer aktif`.

Kontras dengan aplikasi yang di “load”, eksekusi dan terminasi .

Operasi Sistem Komputer

Sistem komputer modern yang umum terdiri dari satu atau lebih CPU dan sejumlah pengendali perangkat terhubung melalui sebuah bus umum yang menyediakan akses ke memori

Sejarah Perkembangan Sistem Operasi

Generasi I (1940-1955)

Awal pengembangan sistem komputasi elektronik, mengganti mesin komputasi mekanis.

Generasi II (1955-1965)

Batch processing unit. Job dikumpulkan dalam satu rangkaian kemudian dieksekusi secara berurutan.

Generasi III (1965-1980)

Multiuser

Multiprogramming

Time Sharing

Spooling

Generasi IV (1980-199x)

Network Operating System

Distributed Operating System

Evolusi Sistem Operasi

OS sederhana

Program tunggal, satu user, satu mesin komputer (CPU) : komputer generasi pertama, awal mesin PCs, controller: lift, Playstation etc.

No problems, no bad people, no bad programs => interaksi sederhana

Problem: terbatas pemakaiannya;

1-Simple Batch System

Memakai seorang operator

User ≠ operator

Menambahkan card reader

Mengurangi waktu setup: batch jobs yang mirip/sama

Automatic job sequencing – secara otomatis kontrol akan di transfer dari satu job ke job yang lain.

Bentuk OS primitif

Resident monitor

Fungsi monitor: awal (initial) melakukan kontrol

Transfer control ke job (pertama)

Setelah job selesai, control kembali ke monitor

Control cards: mengatur batch jobs

2-Multiprogramming Batch Systems

Multiprogramming

I/O rutin dikendalikan dan diatur oleh sistim

Memory management – sistim harus mengalokasikan memori untuk beberapa jobssekaligus

CPU scheduling – sistim harus memilih jobs mana yang telah siap akan dijalankan

Alokasi dari I/O devices untuk jobs dan proteksi bagi I/O devices tersebut

3-Time-Sharing Systems –  
Interactive Computing

CPU melakukan multiplex pada beberapa jobs yang berada di memory (dan disk)

CPU hanya dialokasikan kepada jobs yang telah siap dan berada di memori

Besar memori masih sangat terbatas:

Pada job dilakukan swapped in dan out dari memory ke disk.

Komunikasi on-line (interaktif) antara user dan sistim: jika OS telah selesai mengeksekusi satu perintah, menunggu perintah berikut bukan dari “card reader”, tapi dari terminal user

On-line system harus tersedia bagi user yang akan mengakses data dan kode

4-Parallel Systems (multiprosesor)

Sistim multiprosesor: lebih dari satu CPU yang terhubung secara dekat satu sama lain

Symmetric multiprocessing (SMP)

Setiap prosesor menjalankan “identical copy” dari OS

Banyak proses dapat berjalan serentak murni dengan menggunakan resources pada masing-masing CPU

Banyak modern operating systems mendukung SMP

Parallel Systems (Cont.)

Asymmetric multiprocessing

Setiap prosesor telah ditentukan untuk menjalankan task tertentu

Master processor mengontrol, menjadwalkan dan mengalokasikan task ke slave processors

Banyak digunakan oleh sistemyang besar (main-frame)

5-Distributed Systems

Distribusikan kemampuan komputasi dan “resources” ke berbagai komputer di jaringan.

Loosely coupled system

Setiap prosessor memiliki lokal memori

Komunikasi prosessor satu dengan yang lain melalui beragam jalur komunikasi, contoh : high-speed buses dan jalur telepon.

Distributed Systems (cont)

Manfaat distributed systems.

Resources Sharing

Waktu komputasi cepat– load sharing

Reliability

Komunikasi

Membutuhkan Infrastruktur jaringan.

Local Area Networks (LAN) atau Wide Area Networks (WAN)

Sistem bisa berbentuk client-server atau peer-to-peer .

Struktur Umum Client-Server

6-Clustered Systems

Clustering memungkinkan dua atau lebih sistem melakukan share strorage

Memiliki realibilitas yang tinggi.

Asymmetric clustering: satu server menjalankan aplikasi sementara server lain dalam keadaan standby.

Symmetric clustering: semua N host menjalankan aplikasi.

7-Real-Time Systems

Digunakan sebagai control device untuk aplikasi khusus (misalkan medical imaging systems, industrial control process dll).

Kemampuan untuk beroperasi, response dalam batasan “waktu tertentu” => OS harus sederhana, cepat, dan dapat memenuhi jadwal task (scheduling dll).

Real-Time Systems (Cont.)

Hard real-time system.

Secondary storage sangat terbatas atau tidak ada (menggunakan ROM, flash RAM).

Task dapat diprediksi/ditentukan: waktu selesai dan response.

Soft real-time system

Lebih leluasa batasan waktu dari “hard real-time system”.

Lebih umum digunakan di industri, aplikasi multimedia (video streaming, virtual reality).

8-Handheld Systems

Personal Digital Assistants (PDAs)

Telepon seluler

Issues:

Memori yang terbatas

Prosessor yang lambat

Display screen yang kecil.

Migrasi Sistem Operasi vs.   
Sistem Komputer

Lingkungan Komputasi

Komputasi Tradisional

Komputasi berbasi Web (Web-Based Computing)

Komputasi pada Embedded System (Embedded Computing)

# Struktur system komputer

Struktur   
Sistem Komputer

Arsitektur Sistem Komputer

🟑 Operasi Sistem Komputer

CPU devices dan I/O dapat beroperasi secara serentak (concurrent)

Efisiensi pemakaian CPU

Semua request ke I/O dikendalikan oleh I/O systems:

Setiap device terdapat controller yang mengendalikan device tertentu, misalkan video display => video card, disk => disk controller.

Setiap device controller mempunyai local buffer.

CPU memindahkan data dari/ke memory ke/dari local buffer.

Setelah itu controller akan mengirimkan data dari buffer ke device.

Bagaimana mekanisme I/O supaya CPU dapat melakukan switch dari satu job ke job lain?

Operasi Sistem Komputer (Cont.)

Ilustrasi:

Instruksi CPU dalam orde: beberapa mikro-detik

Operasi read/write dari disk: 10 – 15 mili-detik

Ratio: CPU ribuan kali lebih cepat dari operasi I/O

Jika CPU harus menunggu (idle) sampai data transfer selesai, maka utilisasi CPU sangat rendah (lebih kecil 1%).

Solusi: operasi CPU dan I/O harus overlap

Concurrent: CPU dapat menjalankan beberapa I/O device sekaligus

CPU tidak menunggu sampai operasi I/O selesai tapi melanjutkan tugas yang lain

Bagaimana CPU mengetahui I/O telah selesai?

Programmed I/O (1)

Programmed I/O

Mekanisme CPU yang bertanggung jawab memindahkan data dari/ke memori ke/dari controller

CPU bertanggung jawab untuk jenis operasi I/O

Transfer data dari/ke buffer

Controller melakukan detil operasi I/O

Jika telah selesai memberikan informasi ke CPU => flag

Bagaimana CPU mengetahui operasi telah selesai?

Apakah menguji flag? Seberapa sering?

Programmed I/O (2)

CPU harus mengetahui jika I/O telah selesai => hardware flag (controller)

Polling: CPU secara periodik menguji flag (true or false)

Menggunakan instruksi khusus untuk menguji flag

Masalah: seberapa sering? “wasted CPU time !”? Antar I/O device berbeda “speed”!

Interrupt:

Bantuan hardware – melakukan interupsi pada CPU jika flag tersebut telah di-set (operasi I/O telah selesai)

Interrupt

Interrupt:

CPU transfer control ke “interrupt service routine”, => address dari service routine yang diperlukan untuk device tsb.

Interrupt handler: menentukan aksi/service yang diperlukan

Struktur interrupt harus menyimpan address dari instruksi yang sedang dikerjakan oleh CPU (interrupted).

CPU dapat resume ke lokasi tersebut jika service routine telah selesai dikerjakan

Selama CPU melakukan service interrupt, maka interrupt selanjutnya tidak akan dilayani “disabled”, karena CPU tidak dapat melayani interrupt (lost).

Pengoperasian sistem tersebut menggunakan *interrupt* driven.

Interrupt Handling

Hardware dapat membedakan devices mana yang melakukan interupsi.

Jenis interupsi :

*polling*

*vectored* interrupt system

Tugas sistim operasi menyimpan status CPU (program counter, register dll)

Jika service routine telah selesai => CPU dapat melanjutkan instruksi terakhir yang dikerjakan

Sistim operasi akan “load” kembali status CPU tersebut.

🟑 Struktur I/O

User request I/O:

CPU: load instruksi ke register controller

Controller: menjalankan instruksi

Setelah I/O mulai, control kembali ke user program jika operasi I/O telah selesai

Instruksi khusus: wait => CPU menunggu sampai ada interrupt berikutnya dari I/O tersebut.

Paling banyak hanya mempunyai satu I/O request.

Keuntungan: CPU mengetahui secara pasti device mana yang melakukan interrupt (operasi I/O selesai).

Kerugian: operasi I/O tidak dapat serentak untuksemua device

I/O Interrupt

Pilihan lebih baik: asynchronous I/O

Setelah I/O mulai, kendali langsung kembali ke user program tanpa menunggu I/O selesai

CPU dapat melanjutkan operasi I/O untuk device yang lain

User program dapat menjalankan program tanpa menunggu atau harus menunggu sampai I/O selesai.

*System call* – request ke OS untuk operasi I/O dan menunggu sampai I/O selesai.

Potensi lebih dari satu device

User hanya dapat menggunakan I/O melalui system call

*Device-status table* memuat informasi untuk setiap I/O device: tipe, alamat, status dll

OS mengatur tabel ini dan mengubah isinya sesuai dengan status device (interrupt)

Dua Metode I/O

Device-Status Table

Direct Memory Access (DMA)

Jika I/O devices sangat cepat (“high-speed”),beban CPU menjadi besar harus mengawasi transfer data dari controller ke memory dan sebaliknya.

Hardware tambahan => DMA controller dapat memindahkan blok data dari buffer langsung ke memory tanpa menggangu CPU.

CPU menentukan lokasi memory dan jika DMA controller telah selesai => interrupt ke CPU

Hanya satu interrupt ke CPU untuk sekumpulan data (blok).

🟑 Struktur Storage

Main memory

Media penyimpanan, dimana CPU dapat melakukan akses secara langsung

Secondary storage

Tambahan dari main memory yang memiliki kapasitas besar dan bersifat nonvolatile

Magnetic disks

Metal keras atau piringan yang terbungkus material magnetik

Permukaan disk terbagi secara logikal dalam *track*, yang masing-masing terbagi lagi dalam *sector*

Disk controller menentukan interaksi logikal antara device dan komputer

Mekanisme Pergerakan Head-Disk

Hirarki Storage

Hirarki sistem storage, diorganisasikan dalam bentuk :

Kecepatan

Biaya

Volatilitas

*Caching*

Penduplikasian informasi ke dalam sistem storage yang cepat dapat dilakukan melalui cache pada secondary storage

Hirarki Storage-Device

Caching

Menggunakan memori berkecepatan tinggi untuk menangani akses data saat itu juga (yang terbaru)

Membutuhkan manajemen cache.

Caching mengenalkan tingkatan lain dalam hirarki storage, dimana data secara serentak disimpan pada lebih dari satu tingkatan secara konsisten

Migrasi dari Disk ke Register

🟑 Proteksi Hardware

Dual-Mode Operation

Proteksi I/O

Proteksi Memory

Proteksi CPU

Dual-Mode Operation

Penggunaan resource sharing membutuhkan sistem operasi yang menjamin suatu program yang salah tidak menyebabkan program lain tidak terpengaruh

Menyediakan dukungan hardware yang dibedakan ke dalam dua mode operasi :

1. *User mode* – eksekusi dilakukan untuk kepentingan user.

2. *Monitor mode* (disebut juga *kernel mode* atau *system mode*) – eksekusi dilakukan untuk kepentingan sistem operasi.

Dual-Mode Operation (Cont.)

*Mode bit* ditambahkan pada computer hardware (CPU) untuk indikasi mode sekarang: monitor (0) atau user (1).

Jika terjadi interrupt/fault/error => hardware mengubah mode ke monitor

Proteksi I/O

Semua instruksi I/O adalah instruksi privileged:

Hanya dapat dilakukan melalui OS

OS dapat mencegah “request” ke I/O dengan melihat mode saat ini.

OS menjaga supaya program user tidak dapat menjadi “monitor mode” untuk mencegah user program melakukan:

Menangani interrupt: dengan mengubah alamat interrupt vector.

Mengubah status dan data pada “device table”

Penggunaan System Call untuk Pengoperasian I/O

Proteksi Memory

Melindungi memori terutama untuk isi:

interrupt vector dan interrupt service routines.

Cara umum adalah setiap user program hanya dapat mengakses lokasi memori yang telah dibatasi (disediakan untuk program tsb).

Range address – alamat yang valid

Base register – menyimpan alamat terkecil memori secara fisik

Limit register – besarnya jangkauan memori yang diijinkan

Memori diluar range tersebut tidak dapat diakses oleh user program tsb.

Penggunaan Base dan Limit Register

Proteksi Alamat Hardware

# Manajemen Proses

* Manajemen Proses

Pertemuan 3

* Proses
* Program yang sedang dieksekusi
  + Istilah pada buku teks: *job, task* dan *process* (dapat diartikan sama)
* Proses tidak hanya sekedar suatu kode program (*text section*), melainkan meliputi beberapa aktivitas yang bersangkutan seperti:
  + Melacak posisi instruksi (sequential execution): program counter
  + Menyimpan data sementara var., parameter, return value: stack
  + Menyimpan data (initial, global variable dll): data section
  + Menyimpan status proses (contoh, aktif, wait I/O request dll.)
* Proses (cont.)
* Proses adalah sebuah program dieksekusi yang mencakup *program counter*, register, dan variabel di dalamnya.
  + Aktif (proses=>memori) vs pasif (program => file)
  + Instruksi pada program (code) akan dieksekusi secara berurut (sekwensial) sesuai dengan “line code” (stored program concept).
* Sistem Operasi mengeksekusi proses dengan dua cara yaitu

*Batch System* yang mengeksekusi *jobs* dan *Time-shared System* yang mengatur pengeksekusian program pengguna (*user*) atau *tasks*.

* Proses (cont.)
* Sistem operasi *UNIX* mempunyai *system call fork* yang berfungsi untuk membuat proses baru
* Proses yang memanggil *system call fork* ini akan dibagi jadi dua, proses induk dan proses turunan yang identik.
* Pembuatan Proses
* Umumnya proses dapat membuat proses baru (child process).
  + Child process dapat membuat proses baru. Induk->anak->cucu
  + Terbentuk “tree” dari proses.
* Pilihan hubungan antara parent dan child proses:
  + Resource sharing
    - Parent dan child berbagi resource
    - Children berbagi subset dari resource milik parents.
    - Parent dan child tidak berbagi resource.
  + Execution
    - Parent dan children melakukan eksekusi secara serempak.
    - Parent menunggu hingga children selesai.
* Pembuatan Proses (Cont.)
* Address space
  + Child menduplikasi parent.
  + Child memiliki program yang di load ke dalamnya.
* Contoh UNIX :
  + fork system call membuat proses baru
  + execve (EXEC) :
    - menjalankan program spesifik yang lain
    - nama program tersebut menjadi parameter dari system call
    - EXEC (sering di load sesudah menjalankan fork).
  + Tahapan pembuatan proses baru:
    - Periksa apakah masih terdapat ruang pada PCB.
    - Mencoba mengalokasikan memori untuk proses baru.
    - Mengisi informasi untuk proses baru: nama proses, id, copy data dari parent dll.
    - Mencantumkan informasi proses ke kernel OS.
* Terminasi Proses
* Suatu proses diterminasi ketika proses tersebut telah selesai mengeksekusi perintah terakhir serta meminta sistem operasi untuk menghapus perintah tersebut dengan menggunakan *system call* exit.
* Proses dapat mengembalikan data keluaran kepada proses induk-nya melalui *system call* wait
* Terminasi Proses
* Proses dapat berakhir:
  + Eksekusi instruksi terakhir (atau keluar: exit system call).
  + OS yang akan melakukan dealokasi (memory, file resources).
* UNIX (MINIX):
  + Output signal dari child ke parent
  + Jika parent tidak menunggu (via wait system call), proses akan terminate tapi belum di release dari PCB (status: ZOMBIE).
  + Proses dengan status ZOMBIE (parent telah terminate), akan menjadi child dari proses “init”.
* Parent dapat menghentikan eksekusi proses child secara paksa.
  + Parent dapat mengirim signal (abort, kill system call).
* Status Proses
* Saat proses dijalankan (executed) maka status dari proses akan berubah
  + Status proses tidak selamanya aktif menggunakan CPU).
  + Sering proses menunggu I/O complete => status wait, sebaiknya CPU diberikan kepada proses yang lain.
  + Mendukung multi-tasking – utilisasi CPU dan I/O
* Diagram Status Proses
* Penjelasan gambar
* New
  + Masih dalam tahap inisiasi oleh prosedur
  + Meliputi alokasi memory utama untuk proses
  + Pengisian tabel proses
  + Pembuatan struktur data kendali untuk menyimpan informasi dan status proses
  + Belum siap untuk di eksekusi
  + Kondisi yang memicu proses new
    - Login ke sistem operasi
    - Permintaaan eksekusi program
    - Aplikasi yang menciptakan proses anak
    - Penciptaan proses baru dari eksekusi batch
* Penjelasan gambar
* Ready
  + Proses yang telah berhasil di inisiasi
  + Antrian penjadwalan prosesor dengan cara menyisipkan proses baru ke dalam antrian
  + Berisi referensi atau pointer ke struktur data kendali proses
  + Menandakan suatu proses siap berkompetisi untuk mendapatkan alokasi prosesor
  + Schduler adalah sistem operasi yang bertugas untuk memilih proses yang berada dalam proses ready
* Penjelasan gambar
* Running
  + Proses menguasai prosesor sepenuhnya
  + Memiliki tiga kemungkinan
    - Terminated, proses yang telah selesai
    - Ready, jika jatah waktu yang dialokasikan sudah habis
    - Blocked/waiting
    - Penjelasan gambar
* Blocked/Waiting
  + Proses membutuhkan pembacaan data dari piranti I/O
  + Proses ini akan disisipkan pada antrian penjadwalan peranti I/O atau event
  + Jika I/O yang di tunggu sudah selesai maka proses akan kembali ke antrian ready dan menunggu pemillihan oleh schedule
  + Penjelasan gambar
* EXIT/Terminated
  + Proses tersebut sudah dihentikan eksekusinya
    - Proses telah selesai secara normal
    - Batas waktu total sudah terlewati
    - Kekurang ruang memory
    - Pelanggaran batas memory
    - Pelanggaran proteksi berkas
    - Kesalahan aritmatika
    - Waktu tunggu melebihi batas
    - Terjadi kegagalan I/O
    - Instruksi tidak benar
    - Terjadi pemakaian instruksi yang tidak di izinkan
* Siklus Hidup Proses
* Penjelasan gambar
* keadaan proses blocked dapat berlangsung lama dan ruang memory utama tidak dapat digunakan oleh proses yang lain
* Untuk mengatasi hal itu maka dibuat status suspended blocked
* Penundaan eksekusi proses yang bersatutus blocked yang dilakukan oleh sistem ataupun proses lain akan menyebabkan status proses beralih menjadi suspendedblocked
* Informasi Proses

Dimanakah informasi proses disimpan?

* Data struktur dari OS dalam bentuk table :
  + Satu entry table/linked list => struktur data untuk menampung informasi satu proses (array of structure).
  + Setiap entry pada tabel proses menyimpan satu proses. Contoh: MINIX (src/kernel/proc.h) => struct proc { … };
* Informasi yang disimpan:
  + Informasi internal CPU: isi register-register, program counter, status CPU dll (umumnya dalam bentuk stack frame).
  + Identifikasi proses: nama proses, proses number/index, proses id.
  + Accounting dan timer: user time, system time, alarm etc.
  + Resources: memory & file management.
* Process Control Block (PCB)
* CPU Switch Dari Satu Proses ke Proses Lainnya
* Penjadwalan Proses
* Apakah tujuan dari multiprogramming?
  + “Maximize” pemakaian CPU secara efisien (jadwal dan giliran pemakaian CPU).

=> CPU digunakan oleh proses-proses terus menerus

* Apakah tujuan dari “time-sharing”?
  + Pemakaian CPU dapat di switch dari satu proses ke proses lain (concurrent process execution)

=> sesering mungkin, user dapat berinteraksi dengan sistim

* Bagaimana jika sistim prosesor tunggal?
  + “Hanya ada satu proses yang dapat dijalankan”
  + Proses lain menunggu sampai CPU dapat dijadwalkan (schedule) ke proses tsb
* Penjadwalan Proses
* Proses dapat berubah status dan berpindah dari satu antrian ke antrian yang lain
  + Proses dengan status “ready” berada di ReadyQueue
    - Menunggu giliran/dipilih oleh scheduler => menggunakan CPU
  + Selama eksekusi (status “run”) events yang dapat terjadi:
    - I/O request => I/O wait berada pada DeviceQueue
    - Create “child” proses => Jalankan proses “child”, tunggu sampai proses selesai (wait)
    - Time slice expired => Waktu pemakaian CPU habis, interrupt oleh scheduler, proses akan berpindah ke ReadyQueue
    - Representasi Penjadualan Proses
* Penjadual / Schedulers
* Bagaimana schedulers memilih proses atau program (decision)?
  + Lebih dari satu proses atau program yang akan dijalankan?
* Long-term scheduler (or job scheduler) – memilih proses/program yang mana yang akan di load dan berada di ready queue.
  + Kemungkinan terdapat proses atau job baru.
  + Kemungkinan proses dipindahkan dari memori ke disk (swap out).
* Short-term scheduler (or CPU scheduler) – memilih proses yang mana yang berada di ready queue akan “run” (mendapatkan jatah CPU).
* Alih Konteks / Context Switch
* Jika Scheduler switch ke proses lain, maka sistim harus menyimpan “informasi” proses sekarang (supaya dapat dijalankan kembali)
* Load “informasi” dari proses baru yang berada di PCB
* Waktu Context-switch adalah overhead; sistem tidak melakukan pekerjaan saat terjadi switch.
  + Sangat tergantung pada waktu di hardware
  + OS modern mencari solusi untuk mengurangi overhead waktu switch proses
* Kerjasama Proses
* Proses independent tidak mempengaruhi eksekusi proses yang lain
* Kerjasama proses dapat mempengaruhi atau dipengaruhi oleh eksekusi proses yang lain
* Keuntungan kerjasama proses :
  + Sharing informasi
  + Meningkatkan kecepatan komputasi
  + Modularitas
  + Kemudahan

# Thread

* Thread
* Materi
* Pengenalan Thread
* Pemrograman Multicore
* Model Multithread
* Thread Library
* Implicit Threading
* Permasalahan dalam Thread
* Thread dalam Berbagai Sistem Operasi
* Tujuan
* Memperkenalkan konsep thread – unit dasar dari penggunaan CPU yang terbentuk dari sistem multithread
* Mendiskusikan API untuk library thread Pthreads, Windows, dan Java
* Mengeksplor beberapa strategi implicit threading
* Membahas masalah pada program multithread
* Pengenalan Thread
* Thread
* Unit dasar dari pemanfaatan CPU
* Pada dasarnya satu proses terdiri atas satu thread (single-thread)
  + Dalam sekali waktu hanya ada satu tugas yang bisa dikerjakan dalam satu proses
* Padahal program-program terkini butuh menjalankan beberapa tugas dalam sekali waktu.
  + Butuh multi-thread dalam satu proses.
* Contoh Kebutuhan Multithreading
* Web browser
  + Thread untuk menampilkan halaman web, gambar
  + Thread untuk mengunduh data dari jaringan
* Pengolah kata
  + Thread untuk menerima dan menampilkan hasil pengetikan
  + Thread untuk mengecek grammar
* Webserver
  + Thread untuk menangani request dari banyak user
  + Contoh Multithread pada Webserver
* Question

Mengapa harus multi-thread? Mengapa tidak pakai multiprocess saja? Apa keuntungan pakai multithread?

* Keuntungan Multithread
* Responsiveness
  + Program tetap bisa berjalan walaupun ada sebagian tugas yang memakan waktu lama.
  + Contoh : Web browser tetap dapat digunakan walaupun sedang ada aktifitas download.
* Resource sharing
  + Multiprocess : Butuh implementasi shared memory atau message passing untuk berbagi data antar-proses.
  + Multithread : Antar-thread dalam satu proses sudah berbagi data
* Economy
  + Multiprocess : Butuh alokasi banyak memori dan resource untuk membuat proses baru
  + Multithread : Antar-thread sudah berbagi resource yang dialokasikan untuk sebuah proses
* Scalability
  + Process : Satu processor hanya bisa mengeksekusi satu proses
  + Thread : Setiap thread dalam satu proses dapat dieksekusi secara paralel oleh processor yang berbeda
* Single Thread vs Multi Thread
* Multicore Program
* Multicore Program
* Multicore/Multiprocessor : ada lebih dari satu processor/core processor dalam satu system
* Pemrograman multithread memungkinkan pemanfaatan fitur multicore pada suatu system
* Contoh sebuah program dengan 4 threads
  + Single core :
    - Eksekusi thread dilakukan secara bergantian (concurrent)
  + Multi Core :
    - Eksekusi dilakukan secara parallel sesuai jumlah core
    - Memungkinkan pembagian tugas dalam satu program
* Question
* Apa perbedaan concurrent dan parrarel?
* Concurrency vs. Parallelism
* Concurrent execution pada single core
  + Hanya ada satu thread yang dieksekusi sekali waktu
  + Thread dieksekusi dengan progress tertentu secara bergantian
* Paralleilism pada multi-core
  + Beberapa thread dieksekusi secara parallel sesuai jumlah core
  + Tantangan Pemrograman pada Sistem Multi-core
* Identifikasi aktifitas
  + Identifikasi bagian program mana saja yang dapat berjalan terpisah
* Keseimbangan
  + Tugas yang dipisah memiliki beban yang seimbang
* Pemisahan data
  + Bagaimana membagi data antar bagian program yang terpisah?
* Ketergantungan data
  + Bagaimana jika data pada satu tugas bergantung dengan data dari tugas lain?
* Testing dan debugging
  + Bagaimana melakukan testing dan debugging pada bagian program yang terpisah?
* Tipe Parralelism
* Data parralelism
  + Task parralelism
* Data Parallelism
* Distribusi bagian-bagian data ke masing-masing core processor berbeda untuk diolah
* Operasi pada masing-masing core sama
* Contoh :
  + Operasi penjumlahan bilangan 1 sampai N
  + Ada 2 core processor
  + Penjumlahan data 1 sampai (N/2) di core 1 dan (N/2+1) sampai N di core 2
* Task Parallelism
* Distribusi tugas (thread) berbeda ke masing-masing core processor yang berbeda
* Operasi pada masing-masing core beda
* Contoh :
  + Core 1 menghitung standard deviasi
  + Core 2 mencari nilai max
* Multithreading Model
* User Thread vs Kernel Thread
* Berdasarkan pengendalinya, thread dibagi :
  + User thread
  + Kernel thread
* User thread dikendalikan oleh program pada level user tanpa campur tangan kernel
* Kernel thread dikendalikan langsung oleh sistem operasi
* Ada beberapa model hubungan antara kernel dan user thread
* Model Many-to-One
* Beberapa user-thread dipetakan ke satu kernel-thread
* Kelebihan
  + Tidak ada batasan jumlah user-thread yang bisa dibuat
* Kelemahan
  + Satu thread melakukan blocking (misal I/O), seluruh proses ter-block
  + Multithread tidak dapat dijalankan secara parallel pada sistem multicore
* Contoh
  + Solaris Green Threads
  + GNU Portable Threads
* Model One-to-One
* Satu user-thread dipetakan ke satu kernel-thread
* Pembuatan satu user-thread = Pembuatan satu kernel-thread
* Kelebihan
  + Satu thread blocking tidak akan berpengaruh ke thread lain
  + Multithread dapat dijalankan secara parallel pada sistem multicore
* Kelemahan
  + Jumlah user-thread yang bisa dibuat terbatas
    - Pembuatan kernel-thread dapat membebani performa
* Contoh : Windows, Linux
* Model Many-to-Many
* Beberapa user-thread dipetakan ke beberapa kernel-thread (multiplexing)
* Jumlah kernel-thread yang dibuat bergantung aplikasi dan spesifikasi mesin
* Kelebihan :
  + Satu thread blocking tidak akan berpengaruh ke thread lain
  + Tidak ada batasan jumlah user-thread
* Two Level Model
* Mirip M:M, tapi mengijinkan user-level thread dipetakan ke satu kernel thread
* Contoh :
  + IRIX
  + HP-UX
  + TRU64-Unix
  + Solaris 8 ke bawah

# Penjadwalan CPU

Penjadwalan CPU

Penjadual CPU

Algoritma scheduling:

Memilih dari proses-proses yang berada di memori (ready to execute) dan memberikan jatah CPU ke salah satu proses tersebut.

Kapan keputusan untuk algoritma dilakukan:

Saat suatu proses:

1.Switch dari status running ke waiting.

2.Switch dari status running ke ready.

3.Switch dari status waiting ke ready.

4.Terminates.

Penjadualan 1 dan 4 termasuk nonpreemptive

Penjaudualan lainnya termasuk preemptive

Jenis Penjadualan

Preemptive: OS dapat mengambil (secara interrupt, preempt) CPU dari satu proses setiap saat.

Non-preemptive: setiap proses secara sukarela (berkala) memberikan CPU ke OS.

Contoh:

Penjadualan untuk switch dari running ke wait atau terminate: *non-preemptive*.

Penjadualan proses dari running ke ready: *pre-emptive.*

*Prasyarat untuk OS real-time system.*

Dispatcher

Modul Dispatcher: mengatur dan memberikan kontrol CPU kepada proses yang dipilih oleh “short-term scheduler”:

switching context

switching ke user mode

Melompat ke lokasi yang lebih tepat dari user program untuk memulai kembali program

*Dispatch latency* – terdapat waktu yang terbuang (CPU idle) dimana dispatcher menghentikan satu proses dan menjalankan proses lain.

Save (proses lama) dan restrore (proses baru).

Kriteria Penjadualan

Utilisasi CPU: menjadikan CPU terus menerus sibuk (menggunakan CPU semaksimal mungkin).

Throughput: maksimalkan jumlah proses yang selesai dijalankan (per satuan waktu).

Turn around time: minimalkan waktu selesai eksekusi suatu proses (sejak di submit sampai selesai).

Waiting time: minimalkan waktu tunggu proses (jumlah waktu yang dihabiskan menunggu di ready queue).

Response time: minimalkan waktu response dari sistim terhadap user (interaktif, time-sharing system), sehingga interaksi dapat berlangsung dengan cepat.

Kriteria Penjadualan yang Optimal

Memaksimumkan utilisasi CPU

Memaksimumkan throughput

Meminimukan turnaround time

Meminimumkan waiting time

Meminimumkan response time

Algoritma Penjadualan

First-come, first-served (FCFS)

Shortest-Job-First (SJF)

Round-Robin (RR)

First-Come, First-Served (FCFS)

Algoritma:

Proses yang request CPU pertama kali akan mendapatkan jatah CPU.

Sederhana – algoritma maupun struktur data: menggunakan FIFO queue (ready queue).

FIFO: Non preemptive

Timbul masalah “waiting time” terlalu lama jikadidahului oleh proses yang waktu selesainya lama.

Tidak cocok untuk time-sharing systems.

Digunakan pada OS dengan orientasi batch job.

FCFS (Cont.)

Example: Process Burst Time

P1 24

P2 3

P3 3

Diketahui proses yang tiba adalah P1, P2, P3. Gant chart-nya adalah :

Waiting time untuk *P1* = 0; *P2* = 24; *P3* = 27

Average waiting time (waktu tunggu): (0 + 24 + 27)/3 = 17

Average Turn Around time (waktu penyelesaian): (24+27+30)/3 = 27

FCFS (Cont.)

Diketahui proses yang tiba adalah P2, P3, P1. Gant chart-nya adalah :

Waiting time untuk *P1 =* 6*; P2* = 0*; P3 =* 3

Average waiting time: (6 + 0 + 3)/3 = 3

Lebih baik dari kasus sebelumnya

*Average turn around time?*

*Convoy effect* proses yang pendek diikuti proses yang panjang

Shortest-Job-First (SJF)

Penggabungan setiap proses merupakan panjang dari burst CPU berikutnya. Panjang tersebut digunakan untuk penjadualan proses pada waktu terpendek

Terdapat 2 skema :

*nonpreemptive* – CPU hanya satu kali diberikan pada suatu proses, maka proses tersebut tetap akan memakai CPU hingga proses tersebut melepaskannya

*preemptive* –jika suatu proses tiba dengan panjang CPU burst lebih kecil dari waktu yang tersisa pada ekseksusi proses yang sedang berlangsung, maka dijalankan preemtive. Skema ini dikenal dengan Shortest-Remaining-Time-First (SRTF).

SJF akan optimal, keteika rata-rata waktu tunggu minimum untuk set proses yang diberikan

Ketika waktu tiba nya sama maka yang diperhitungkan adalah lama proses (burst time)

Contoh Preemptive SJF

Process Arrival Time Burst Time

*P1* 0.0 7

*P2* 2.0 4

*P3* 4.0 1

*P4* 5.0 4

SJF (preemptive)

Average waiting time = ??

*Average Arround Time = ??*

Round Robin (RR)

Setiap proses mendapat jatah waktu CPU (time slice/quantum) tertentu misalkan 10 atau 100 milidetik.

Setelah waktu tersebut maka proses akan di-preempt dan dipindahkan ke ready queue.

Adil dan sederhana.

Jika terdapat n proses di “ready queue” dan waktu quantum q (milidetik), maka:

Maka setiap proses akan mendapatkan 1/n dari waktu CPU.

Proses tidak akan menunggu lebih lama dari: (n-1) q time units.

Performance

*q* besar ⇒ FIFO

*q* kecil ⇒ *q* harus lebih besar dengan mengacu pada context switch, jika tidak overhead akan terlalu besar

Contoh RR (Q= 20)

Process Burst Time

*P1* 53

*P2* 17

*P3* 68

*P4* 24

Gantt Chart

Tipikal: lebih lama waktu rata-rata turnaround dibandingkan SJF, tapi mempunyai response terhadap user lebih cepat.

Waktu Kuantum dan Waktu Context Switch

Latihan

Jika diketahui 5 antrian proses yaitu: A-B-C-D-E dengan waktu kedatangan semuanya 0, lama proses berturut-turut antara lain 5,2,6,8,3. Hitung rata-rata waktu tunggu (waiting time) dan rata-rata waktu penyelesaiannya (turn around time) menggunakan algoritma FIFO dan SJF preemptive!

Hitung rata-rata waktu tunggu (waiting time) dan rata-rata waktu penyelesaiannya (turn around time) menggunakan algoritma FIFO dan SJF preemptive!

# Sinkronisasi

Sinkronisasi

Overview (1)

*Proteksi OS:*

*Independent* process tidak terpengaruh atau dapat mempengaruhi eksekusi/data proses lain.

“Concurrent Process”

OS: mampu membuat banyak proses pada satu saat

Proses-proses bekerja-sama: sharing data, pembagian task, passing informasi dll

Proses => mempengaruhi proses lain dalam menggunakan data/informasi yang sengaja di-”share”

*Cooperating* process – sekumpulan proses yang dirancang untuk saling bekerja-sama untuk mengerjakan task tertentu.

Overview (2)

Keuntungan kerja-sama antar proses

Information sharing: file, DB => digunakan bersama

Computation speed-up: parallel proses

Modularity: aplikasi besar => dipartisi dalam banyak proses.

Convenience: kumpulan proses => tipikal lingkungan kerja.

“Cooperating Process”

Bagaimana koordinasi antar proses? Akses/Update data

Tujuan program/task: integritas, konsistensi data dapat dijamin

Latar Belakang

Menjamin konsistensi data:

Program/task-task dapat menghasilkan operasi yang benar setiap waktu

Deterministik: untuk input yang sama hasil harus sama (sesuai dengan logika/algroitma program).

Contoh: Producer – Consumer

Dua proses: producer => menghasilkan informasi; consumer => menggunakan informasi

Sharing informasi: buffer => tempat penyimpanan data

*unbounded-buffer,* penempatan tidak pada limit praktis dari ukuran buffer

*bounded-buffer* diasmumsikan terdapat ukuran buffer yang tetap

Bounded Buffer (1)

Implementasi buffer:

IPC: komunikasi antar proses melalui messages membaca/menulis buffer

Shared memory: programmer secara eksplisit melakukan “deklarasi” data yang dapat diakses secara bersama.

Buffer dengan ukuran n => mampu menampung n data

Producer mengisi data buffer => increment “counter” (jumlah data)

Consumer mengambil data buffer => decrement “counter”

Buffer, “counter” => shared data (update oleh 2 proses)

Bounded Buffer (2)

Shared data type *item* = … ;

var *buffer* array

*in*, *out*: 0..*n*-1;

*counter*: 0..*n*;

*in, out, counter* := 0;

Producer process

repeat

…

produce an item in *nextp*

…

while *counter =* n do no-op;

*buffer* [*in*] := *nextp*;

*in* := *in* + 1 mod *n*;

*counter* := *counter* +1;

until false;

Bounded Buffer (3)

Consumer process

repeat

while *counter* = 0 do *no-op*;

*nextc* := *buffer* [*out*];

*out* := *out* + 1 mod *n*;

*counter* := *counter* – 1;

…

consume the item in *nextc*

…

until *false*;

Bounded Buffer (4)

Apakah terdapat jaminan operasi akan benar jika berjalan concurrent?

Misalkan: counter = 5

Producer: counter = counter + 1;

Consumer: counter = counter - 1;

Nilai akhir dari counter?

Operasi concurrent P & C =>

Operasi dari high level language => sekumpulan instruksi mesin: “increment counter”

Load Reg1, Counter

Add Reg1, 1

Store Counter, Reg1

Bounded Buffer (5)

“decrement counter”

Load Reg2, Counter

Subtract Reg2, 1

Store Counter, Reg2

Eksekusi P & C tergantung scheduler (dapat gantian)

T0: Producer : Load Reg1, Counter (Reg1 = 5)

T1: Producer : Add Reg1, 1 (Reg1 = 6)

T2: Consumer: Loag Reg2, Counter (Reg2 = 5)

T3: Consumer: Subtract Reg1, 1 (Reg2 = 4)

T4: Producer: Store Counter, Reg1 (Counter = 6)

T5: Consumer: Store Counter, Reg2 (Counter = 4)

Race Condition

Concurrent C & P

Shared data “counter” dapat berakhir dengan nilai: 4, atau 5, atau 6

Hasilnya dapat salah dan tidak konsisten

Race Condition:

Keadaan dimana lebih dari satu proses meng-update data secara “concurrent” dan hasilnya sangat bergantung dari urutan proses mendapat jatah CPU (run)

Hasilnya tidak menentu dan tidak selalu benar

Mencegah race condition: sinkronisasi proses dalam meng-update shared data

Sinkronisasi

Sinkronisasi:

Koordinasi akses ke shared data, misalkan hanya satu proses yang dapat menggunakah shared var.

Contoh operasi terhadap var. “counter” harus dijamin di-eksekusi dalam satu kesatuan (atomik) :

*counter* := *counter* + 1;

*counter* := *counter* - 1;

Sinkronisasi merupakan “issue” penting dalam rancangan/implementasi OS (shared resources, data, dan multitasking).

Masalah Critical Section

n proses mencoba menggunakan shared data bersamaan

Setiap proses mempunyai “code” yang mengakses/ manipulasi shared data tersebut => “critical section”

Problem: Menjamin jika ada satu proses yang sedang

“eksekusi” pada bagian “critical section” tidak ada proses lain yang diperbolehkan masuk ke “code” critical section dari proses tersebut.

Structure of process *Pi*

Solusi Masalah Critical Section

Ide :

Mencakup pemakaian secara “exclusive” dari shared variable tersebut

Menjamin proses lain dapat menggunakan shared variable tersebut

Solusi “critical section problem” harus memenuhi:

* + 1. Mutual Exclusion: Jika proses Pi sedang “eksekusi” pada bagian “critical section” (dari proses Pi) maka tidak ada proses proses lain dapat “eksekusi” pada bagian critical section dari proses-proses tersebut.
    2. Progress: Jika tidak ada proses sedang eksekusi pada critical section-nya dan jika terdapat lebih dari satu proses lain yang ingin masuk ke critical section, maka pemilihan siapa yang berhak masuk ke critical section tidak dapat ditunda tanpa terbatas.

Solusi (cont.)

* + 1. Bounded Waiting: Terdapat batasan berapa lama suatu proses harus menunggu giliran untuk mengakses “critical section” – jika seandainya proses lain yang diberikan hak akses ke critical section.

Menjamin proses dapat mengakses ke “critical section” (tidak mengalami starvation: proses se-olah berhenti menunggu request akses ke critical section diperbolehkan).

Tidak ada asumsi mengenai kecepatan eksekusi proses proses n tersebut.

Solusi Sederhana : Kasus 2 proses

Hanya 2 proses

Struktur umum dari program code *Pi* dan *Pj*:

Software solution: merancang algoritma program untuk solusi critical section

Proses dapat mengunakan “common var.” untuk menyusun algoritma tsb.

Algoritma 1

Shared variables:

int turn;  
initially turn = 0

turn - i ⇒ *Pi* dapat masuk ke criticalsection

Process *Pi*

do {

while (turn != i) ;

critical section

turn = j;

reminder section

} while (1);

Mutual exclusion terpenuhi, tetapi menentang progress

Algoritma 2

Shared variables

boolean flag[2];  
initially flag [0] = flag [1] = false.

flag [i] = true ⇒ *Pi* siap dimasukkan ke dalam critical section

Process *Pi*

do {

flag[i] := true;  
 while (flag[j]) ; critical section

flag [i] = false;

remainder section

} while (1);

Mutual exclusion terpenuhi tetapi progress belum terpenuhi.

Algoritma 3

Kombinasi shared variables dari algoritma 1 and 2.

Process Pi

do {

flag [i]:= true;  
 turn = j;  
 while (flag [j] and turn = j) ;

critical section

flag [i] = false;

remainder section

} while (1);

Ketiga kebutuhan terpenuhi, solusi masalah critical section pada dua proses

Algoritma Bakery

Critical section untuk n proses

Sebelum proses akan masuk ke dalam “critical section”, maka proses harus mendapatkan “nomor” (tiket).

Proses dengan nomor terkecil berhak masuk ke critical section.

Jika proses *Pi* dan *Pj* menerima nomor yang sama, jika *i* < *j*, maka *Pi* dilayani pertama; jika tidak *Pj* dilayani pertama

Skema penomoran selalu dibuat secara berurutan, misalnya 1,2,3,3,3,3,4,5...

Algoritma Bakery (2)

Notasi <≡ urutan lexicographical (ticket #, process id #)

(*a,b*) < *c,d*) jika *a* < *c* atau jika *a* = *c* and *b* < *d*

max (*a0*,…, *an*-1) dimana a adalah nomor, *k*, seperti pada *k* ≥ *a*i untuk *i* - 0,   
…, *n* – 1

Shared data

var *choosing*: array [0..*n* – 1] of *boolean*

*number*: array [0..*n –* 1] of *integer*,

Initialized: choosing =: *false* ; number => 0

Algoritma Bakery (3)

do {

choosing[i] = true;

number[i] = max(number[0], number[1], …, number [n – 1])+1;

choosing[i] = false;

for (j = 0; j < n; j++) {

while (choosing[j]) ;

while ((number[j] != 0) && (number[j,j] < number[i,i])) ;

}

critical section

number[i] = 0;

remainder section

} while (1);

Sinkronisasi Hardware

Memerlukan dukungan hardware (prosesor)

Dalam bentuk “instruction set” khusus: test-and-set

Menjamin operasi atomik (satu kesatuan): test nilai dan ubah nilai tersebu

Test-and-Set dapat dianalogikan dengan kode:

Test-and-Set (mutual exclusion)

Mutual exclusion dapat diterapkan:

Gunakan shared data,

variabel: *lock: boolean (initially false)*

lock: menjaga critical section

Process Pi:

do {

while (TestAndSet(lock)) ;

critical section

lock = false;

remainder section

}

Semaphore

Perangkat sinkronisasi yang tidak membutuhkan *busy waiting*

Semaphore S – integer variable

Dapat dijamin akses ke var. S oleh dua operasi atomik:

wait (S): while S ≤ 0 do no-op;

S := S – 1;

signal (S): S := S + 1;

Contoh : n proses

Shared variables

var mutex : semaphore

initially mutex = 1

Process Pi

do {  
 wait(mutex);  
 critical section

signal(mutex);  
 remainder section  
} while (1);

Implementasi Semaphore

Didefinisikan sebuah Semaphore dengan sebuah record

typedef struct {

int value;  
 struct process \*L;  
 } semaphore;

Diasumsikan terdapat 2 operasi sederhana :

block menhambat proses yang akan masuk

wakeup(*P*) memulai eksekusi pada proses P yang di block

Implementasi Semaphore (2)

Operasi Semaphore-nya menjadi :

*wait*(*S*):   
 S.value--;

if (S.value < 0) {

add this process to S.L;  
 block;

}

*signal*(*S*):   
 S.value++;

if (S.value <= 0) {

remove a process P from S.L;  
 wakeup(P);

}

Masalah Klasik Sinkronisasi

Bounded-Buffer Problem

Readers and Writers Problem

Dining-Philosophers Problem

Bounded-Buffer Problem

Shared data  
  
semaphore full, empty, mutex;  
  
Initially:  
  
full = 0, empty = n, mutex = 1

Bounded-Buffer Problem : Producer-Consumer

Readers-Writers Problem

Shared data  
  
semaphore mutex, wrt;  
  
Initially  
  
mutex = 1, wrt = 1, readcount = 0

Readers-Writers Problem (2)

Writters Process

wait(wrt);

…

writing is performed

…

signal(wrt);

Dining-Philosophers Problem

Shared data

semaphore chopstick[5];

Semua inisialisasi bernilai 1

Dining-Philosophers Problem

Philosopher *i*:

do {

wait(chopstick[i])

wait(chopstick[(i+1) % 5])

…

eat

…

signal(chopstick[i]);

signal(chopstick[(i+1) % 5]);

…

think

…

} while (1);

Solusi Tingkat Tinggi

Motif:

Operasi wait(S) dan signal(S) tersebar pada code program => manipulasi langsung struktur data semaphore

Bagaimana jika terdapat bantuan dari lingkungan HLL (programming) untuk sinkronisasi ?

Pemrograman tingkat tinggi disediakan sintaks-sintaks khusus untuk menjamin sinkronisasi antar proses, thread

Misalnya:

Monitor & Condition

Conditional Critical Region

Monitor

Monitor mensinkronisasi sejumlah proses:

suatu saat hanya satu yang aktif dalam monitor dan yang lain menunggu

Bagian dari bahasa program (mis. Java).

Tugas compiler menjamin hal tersebut terjadi dengan menerjemahkan ke “low level synchronization” (semphore, instruction set dll)

Cukup dengan statement (deklarasi) suatu section/fungsi adalah monitor => mengharuskan hanya ada satu proses yang berada dalam monitor (section) tsb

Monitor (2)

Monitor (3)

Proses-proses harus disinkronisasikan di dalam monitor:

Memenuhi solusi critical section.

Proses dapat menunggu di dalam monitor.

Mekanisme: terdapat variabel (condition) dimana proses dapat menguji/menunggu sebelum mengakses “critical section”

var *x, y: condition*

Monitor (4)

Condition: memudahkan programmer untuk menulis code pada monitor.

Misalkan : var x: condition ;

Variabel condition hanya dapat dimanipulasi dengan operasi: wait() dan signal()

x.wait() jika dipanggil oleh suatu proses maka proses tsb. akan suspend - sampai ada proses lain yang memanggil: x. signal()

x.signal() hanya akan menjalankan (resume) 1 proses saja yang sedang menunggu (suspend) (tidak ada proses lain yang wait maka tidak berdampak apapun)

Skema Monitor

# Deadlock

* Deadlock
* Masalah Deadlock
* Sekumpulan proses sedang blocked karena setiap proses sedang menunggu (antrian) menggunakan “resources” yang sedang digunakan (hold) oleh proses lain.
* Contoh:
  + OS hanya mempunyai akes ke 2 tape drives.
  + P1 dan P2 memerlukan 2 tape sekaligus untuk mengerjakan task (copy).
  + P1 dan P2 masing-masing hold satu tape drives dan sedang blocked, karena menunggu 1 tape drives “available”.
* Contoh Persimpangan Jalan
* Hanya terdapat satu jalur
* Mobil digambarkan sebagai proses yang sedang menuju sumber daya.
* Untuk mengatasinya beberapa mobil harus *preempt* (mundur)
* Sangat memungkinkan untuk terjadinya *starvation* (kondisi proses tak akan mendapatkan sumber daya).
* Resource-Allocation Graph

Sekumpulan vertex V dan sekumpulan edge E

* V dipartisi ke dalam 2 tipe
  + *P* = {*P*1, *P*2, …, *Pn*}, terdiri dari semua proses dalam sistem.
  + *R* = {*R*1, *R*2, …, *Rm*}, terdiri dari semua sumberdaya dalam sistem
* request edge/permintaan edge : arah edge *P*1 → *Rj*
* assignment edge/penugasan edge – arah edge *Rj* → *Pi*
* Resource-Allocation Graph (Cont.)
* Process
* Resource Type with 4 instances
* *Pi* requests instance of *Rj*
* *Pi* is holding an instance of *Rj*
* Contoh Resource Allocation Graph
* Graf Resource Allocation Dengan Deadlock
* Graf Resource Allocation dengan Cycle Tanpa Deadlock
* Kondisi yang Diperlukan untuk Terjadinya Deadlock
* **Mutual Exclusion**
  + **Serially-shareable resources (mis. Buffer)**
  + Contoh: Critical section mengharuskan mutual exclusion (termasuk resource), sehingga potensi proses akan saling menunggu (blocked).
* **Hold & wait :**
  + **Situasi dimana suatu proses sedang hold suatu resource secara eksklusif dan ia menunggu mendapatkan resource lain (wait).**
* Kondisi yang Diperlukan untuk Terjadinya Deadlock (cont.)
* **No-Preemption Resouce :**
  + Resource yang hanya dapat dibebaskan secara sukarela oleh proses yang telah mendapatkannya
  + Proses tidak dapat dipaksa (pre-empt) untuk melepaskan resource yang sedang di hold
* **Circular wait**
  + Situasi dimana terjadi saling menunggu antara beberapa proses sehingga membentuk waiting chain (circular)
  + Misalkan proses (P0, P1, .. Pn) sedang blok menunggu resources: P0 menunggu P1, P1 menunggu P2, .. dan Pn menunggu P0.
* Metode Penanganan Deadlock
* **Deadlock Prevention**: Pencegahan adanya faktor-faktor penyebab deadlock
* **Deadlock Avoidance**: Menghindari dari situasi yang potensial dapat mengarah menjadi deadlock
* **Deadlock Detection**: Jika deadlock ternyata tidak terhindari maka bagaimana mendeteksi terjadinya deadlock, dilanjutkan dengan penyelamatan (recovery).
* Deadlock Prevention
* Pencegahan: Faktor-faktor penyebab deadlock yang harus dicegah untuk terjadi
* 4 faktor yang harus dipenuhi untuk terjadi deadlock:
  + Mutual Exclusion: pemakaian resources.
  + Hold and Wait: cara menggunakan resources.
  + No preemption resource: otoritas/hak.
  + Circular wait: kondisi saling menunggu.
* Jika salah satu bisa dicegah maka deadlock pasti tidak terjadi!
* Deadlock Prevention (1)
* **Tindakan preventif:**
  + **Batasi pemakaian resources**
  + **Masalah: sistim tidak efisien, tidak feasible**
* **Mutual Exclusion:**
  + tidak diperlukan untuk shareable resources
  + read-only files/data : deadlock dapat dicegah dengan tidak membatasi akses (not mutually exclusive)
  + tapi terdapat resource yang harus mutually exclusive (printer)
* Deadlock Prevention (2)
* **Hold and Wait**
  + Request & alokasi dilakukan saat proses start (dideklarasikan dimuka program)
  + Request hanya bisa dilakukan ketika tidak sedang mengalokasi resource lain; alokasi beberapa resource dilakukan sekaligus dalam satu request
  + Simple tapi resource akan dialokasi walau tidak selamanya digunakan (low utilization) serta beberapa proses bisa mengalami starvation
* Deadlock Prevention (3)
* **Mencegah Circulair Wait**
  + Pencegahan: melakukan total ordering terhadap semua jenis resource
  + Setiap jenis resource mendapatkan index yang unik dengan bilangan natural: 1, 2, . .
    - Contoh: tape drive=1, disk drive=5, printer=12
  + Request resource harus dilakukan pada resource-resource dalam urutan menaik (untuk index sama - request sekaligus)
  + Jika Pi memerlukan Rk yang berindeks lebih kecil dari yang sudah dialokasi maka ia harus melepaskan semua resource Rj yang berindeks >= Rk
* Deadlock Prevention (4)
* **Mencegah No-Preemption**
  + Jika proses telah mengalokasi resource dan ingin mengalokasi resource lain – tapi tidak diperoleh (wait) : maka ia melepaskan semua resource yang telah dialokasi.
  + Proses akan di-restart kelak untuk mecoba kembali mengambil semua resources
* Deadlock Avoidance
* **Pencegahan:**
  + **Apabila di awal proses; OS bisa mengetahui resource mana saja yang akan diperlukan proses.**
  + **OS bisa menentukan penjadwalan yang aman (“safe sequence”) alokasi resources.**
* **Model:**
  + Proses harus menyatakan max. jumlah resources yang diperlukan untuk selesai.
* Algoritma “deadlock-avoidance” secara dinamik akan memeriksa alokasi resource apakah dapat mengarah ke status (keadaan) tidak aman (misalkan terjadi circulair wait condition)
* Jadi OS, tidak akan memberikan resource (walaupun available), kalau dengan pemberian resource ke proses menyebakan tidak aman (unsafe).
* Safe, unsafe , deadlock state
* Safe state
* Prasyarat:
  + Proses harus mengetahui max. resource yang diperlukan (upper bound) => asumsi algoritma.
  + Proses dapat melakukan hold and wait, tapi terbatas pada sekump lan resource yang telah menjadi “kreditnya”.
* Setiap ada permintaan resource, OS harus memeriksa
  + “jika resource diberikan”, dan terjadi “worst case” semua proses melakukan request “max. resource”
  + Terdapat “urutan yang aman” dari resources yang available, untuk diberika ke proses, sehingga tidak terjadi deadlock.
  + Kondisi Safe
* Resources: 12 tape drive.

A (Available): 12 - 10 = 2

Safe sequnce:

2 tape diberikan ke U2,

U2 selesai => Av = 6,

Berikan 3 tape ke U1,

Berikan 3 tape ke U3.

No deadlock.

* Kondisi Unsafe
* Resources: 12 tape drive.

A (Available): 12 - 11 = 1

Terdapat 1 tape available,

sehingga dapat terjadi Deadlock.

* Algoritma Banker’s
* Proses harus “declare” max. kredit resource yang diinginkan.
* Proses dapat block (pending) sampai resource diberikan.
* Banker’s algorithm menjamin sistim dalam keadaan safe state.
* OS menjalankan Algoritma Banker’s,
  + Saat proses melakukan request resource.
  + Saat proses terminate atau release resource yang digunakan => memberikan resource ke proses yang pending request.
  + Algoritma Banker’s (2)
* **Metode :**
  + 1. Scan tabel baris per baris untuk menemukan job yang akan diselesaikan
    2. Tambahkan pada job terakhir dari sumberdaya yang ada dan berikan nomor yang available
  + Ulangi 1 dan 2 hingga :
    1. Tidak ada lagi job yang diselesesaikan (unsafe) atau
    2. Semua job telah selesai (*safe*)
* Algoritma Banker’s (3)
* Misakan terdapat: n proses dan m resources.
* Definisikan:
  + **Available**: Vector/array dengan panjang **m**.

If **available [j] = k**, terdapat **k** instances resouce jenis **Rj** yang dapat digunakan.

* + **Max**: matrix **n x m**.

If **Max [i,j] = k**, maka proses **Pi** dapat request paling banyak k instances resource jenis Rj.

* + **Allocation**: matrix **n x m**.

If **Allocation[i,j] = k** maka **Pi** saat ini sedang menggunakan

(hold) **k** instances Rj.

* + **Need**: matrix **n x m**.

If **Need[i,j] = k**, maka **Pi** palaing banyak akan membutuhkan

instance Rj untuk selesai.

* + **Need [i,j] = Max[i,j] – Allocation [i,j].**
  + Algoritma Safety
* Let *Work* and *Finish* be vectors of length *m* and *n*,respectively. Initialize:

***Work* := *Available // resource yang free***

***Finish* [*i*] = *false* for *i* = 1,3, …, *n.***

* Find and *i* such that both: // penjadwalan alokasi resource

(a) *Finish* [*i*] = *false // asume, proses belum complete*

(b) *Needi* £ *Work // proses dapat selesai, ke step 3*

If no such *i* exists, go to step 4.

* *Work* := *Work* + *Allocation****i*** // proses dapat selesai

*Finish*[*i*] := *true*

go to step 2.

* If *Finish* [*i*] = true for all *i*, then the system is in a safe state.
* Algoritma Safety (2)
* Terdapat 3 proses: n = 3, 1 resource: m = 1
* Jumlah resource m = 12.
* Snapshot pada waktu tertentu:
* Algoritma Safety (3)

Let Need[3]; Max[3]; Aloc[3]; Finish[3]; Work [1];

1. Work = Available; // Work = 2;

Finish[0]=false, Finish[1]=false, Finish[3]=false;

1. do {

FlagNoChange = false;

for I=0 to 2 {

if ((Finish[I] == false)) && (Need[I] <= Work) {

Finis[I] = true;

Work = Work + Aloc[I]; FlagNoChange = true;

}

} until (FlagNoChange);

* Deadlock Detection
* Mencegah dan menghidari dari deadlock sulit dilakukan:
  + Kurang efisien dan utilitas sistim
  + Sulit diterapkan: tidak praktis, boros resources
* Mengizinkan sistim untuk masuk ke “state deadlock”
  + Gunakan algoritma deteksi (jika terjadi deadlock)
    - Deteksi: melihat apakah penjadwalan pemakaian resource yang tersisa masih memungkinkan berada dalam safe state (variasi “safe state”).
  + Skema recovery untuk mengembalikan ke “safe state”
* Single Instance
* Gunakan: resource allocation graph
  + Node mewakili proses, arcus mewakili request dan hold dari resources.
  + Dapat disederhanakan dalam “wait-for-graph”
    - *Pi 🡪* *Pj* if *Pi* is waiting for *Pj.*
  + Secara periodik jalankan algoritma yang mencari cycle pada graph:
    - Jika terdapat siklus (cycle) pada graph maka telah terjadi deadlock.
  + Recovery dari Deadlock
* Batalkan semua proses deadlock
* Batalkan satu proses pada satu waktu hingga siklus deadlock dapat dihilangkan
* Proses mana yang dapat dipilih untuk dibatalkan ?
  + Proses dengan prioritas
  + Proses dengan waktu proses panjang
  + Sumberdaya proses yang telah digunakan
  + Sumberdaya proses yang lengkap
  + Banyak proses yang butuh untuk ditunda
  + Apakah proses tersebut interaktif atau batch
* Recovery dariDeadlock
* Pilih proses – meminimasi biaya
* Rollback – kembali ke state safe, mulai lagi proses dari state tersebut
* Starvation – proses yang sama selalu diambil sebagai pilihan, termasuk rollbak dalam faktor biaya
* Pendekatan Kombinasi
* Kombinasi dari tiga pendekatan dasar
  + prevention
  + avoidance
  + detection
  + Pemisahan sumberdaya ke dalam hirarki kelas