UTS OSu!!!

kita selama 1 semester belajar apa aja ya?
sejujurnya akupun agak ngang ngeng ngong tp aku coba untuk jelasin semampunya ☺

Disusun oleh Amar Fadil

< Kebingunganku thdp os seperti gambar dinosaurus di cover book operating system concepts 10th, kaya, knp harus dinosaurus?!?!

Belajar apa aja sih kira-kira?

- Intro to operating system (definisi, tujuan/goals, komponen, multiprogramming, timesharing, multitasking, interrupt)
- OS Structure (services, structure, kernel, booting, user-kernel mode, syscall, system program)
- Process (concept, scheduling, operations, co-op, IPC, server-client)
- Threads (definisi, models, operations, issues)
- CPU Scheduling (concept, criteria, algorithms [FCFS, SJF, Priority, RR, Multilevel Queue, Multilevel Feedback], thread scheduling, multiprocessor, real-time [Priority, Rate-Monotonic, EDF])
- Synchronizations (race condition, critical section, Peterson's algo, hardware support, mutex lock, semaphore, classic problem, monitors)
- Deadlock (definition, system model, characteristics, RAG, handling: [prevention, avoidance: safety/bankers algo/safe state, detection: algo/recovery])

Intro to OS

Basic definition & goals, component, jobs, multiprogramming, timesharing, multitasking, interrupt

Basic Definition & Goals

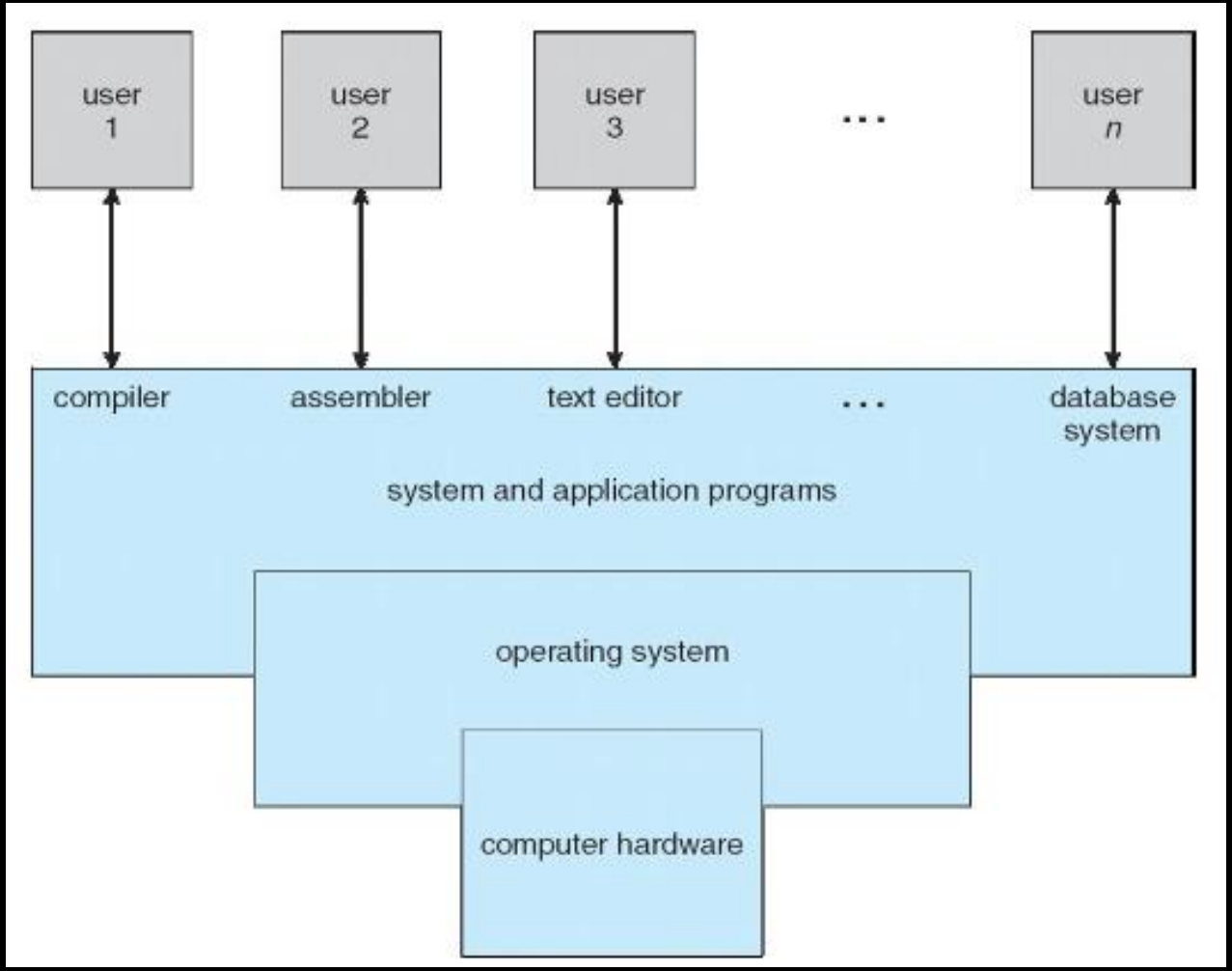
- OS = software yang memanajemen hardware komputer (resource allocator) serta pengontrol program (control program) dan menjembatani antara pengguna dengan hardware komputer [gak ada definisi pastinya sih, ini versi aku].
- Kernel = satu program yang berjalan setiap saat di komputer.
- System Program = program yang biasanya udah ada secara default di operating system, tapi bukan kernel. Contohnya: task manager, disk management, cmd, dkk.
- Application Program = program selain dari yang udah ada di OS (yang dibuat oleh user ataupun diinstall dan dijalankan user). Contohnya: MS Office, WinRAR, dkk.
- Tujuan ada OS:

1. Memudahkan user dalam menggunakan komputer (ease of use)
2. Manajemen & koordinasi antar hardware (resource utilization).
3. Mengalokasikan sebagian hardware untuk setiap proses (resource allocator).
4. Menjalankan program user/sistem (control program).

OS Component

- 4 komponen OS:

1. Hardware: basic computing resources (CPU, memory, I/O, dkk)
2. Operating System/Kernel
3. Application Programs (word processors, games, browser, dkk)
4. Users (manusia, mesin, atau computer lainnya)

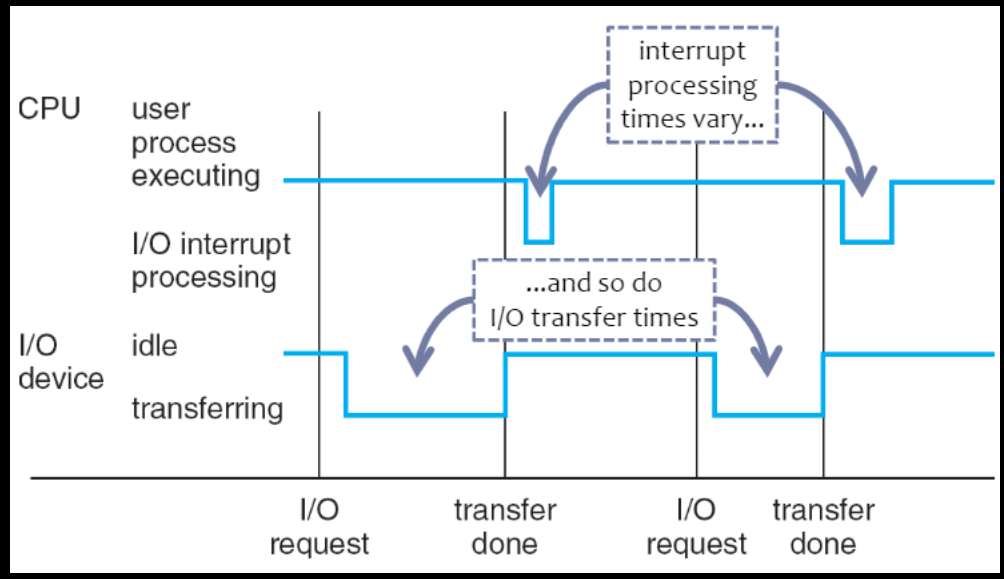


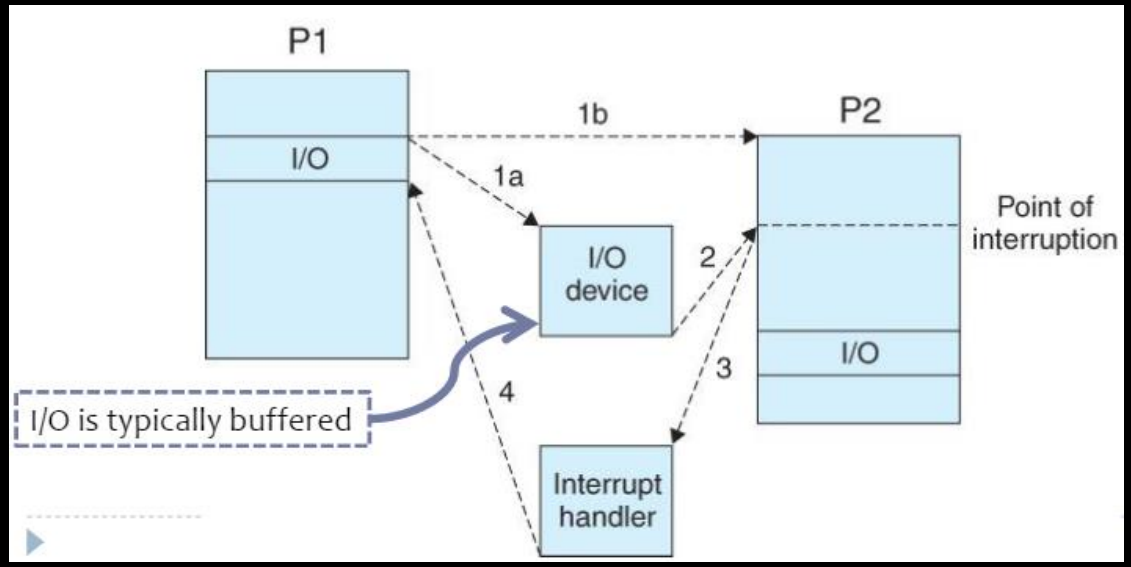
Jobs, Multiprogramming, Timesharing-Multitasking

- Job = suatu unit berupa kerjaan yang harus dilakukan sistem operasi. Tiap job berupa program yang dijalankan beserta input data dari user yang dipakai program.
- Multiprogramming = menjalankan beberapa program (disebut process) dalam satu waktu untuk **meningkatkan utilisasi CPU** dengan mengorganisasi program-program tersebut sehingga selalu terdapat program yang dijalankan CPU.
- Timesharing = Supaya adil, CPU menjalankan job secara berganti-gantian, dengan memberikannya CPU time yang tetap (bisa disingkat jika terblock sama I/O req), kemudian melakukan context switching (pergantian proses yang berjalan).
- Multitasking = CPU dapat menjalankan beberapa program sekaligus dengan switching program dalam frekuensi tinggi, sehingga memberikan user **response time** yang cepat.

Interrupt

- Interrupt = berhenti melakukan kerjaan yang sedang dilakukan CPU dan mengeksekusi fungsi (interrupt handler) sesuai dengan service routine dan kode interrupt yang diterima, kemudian melanjutkan kerjaannya.
- Interrupt biasanya ditrigger oleh hardware event (mouse movement, keystroke, etc).
- Software dapat melakukan interrupt dengan mengeksekusi **system call** (graphics output, disk read request). Biasanya ini disebut sebagai **software interrupt**.
- Jika terdapat kesalahan (exception), software dapat mengirimkan **trap** yang akan menginvoke suatu system call.

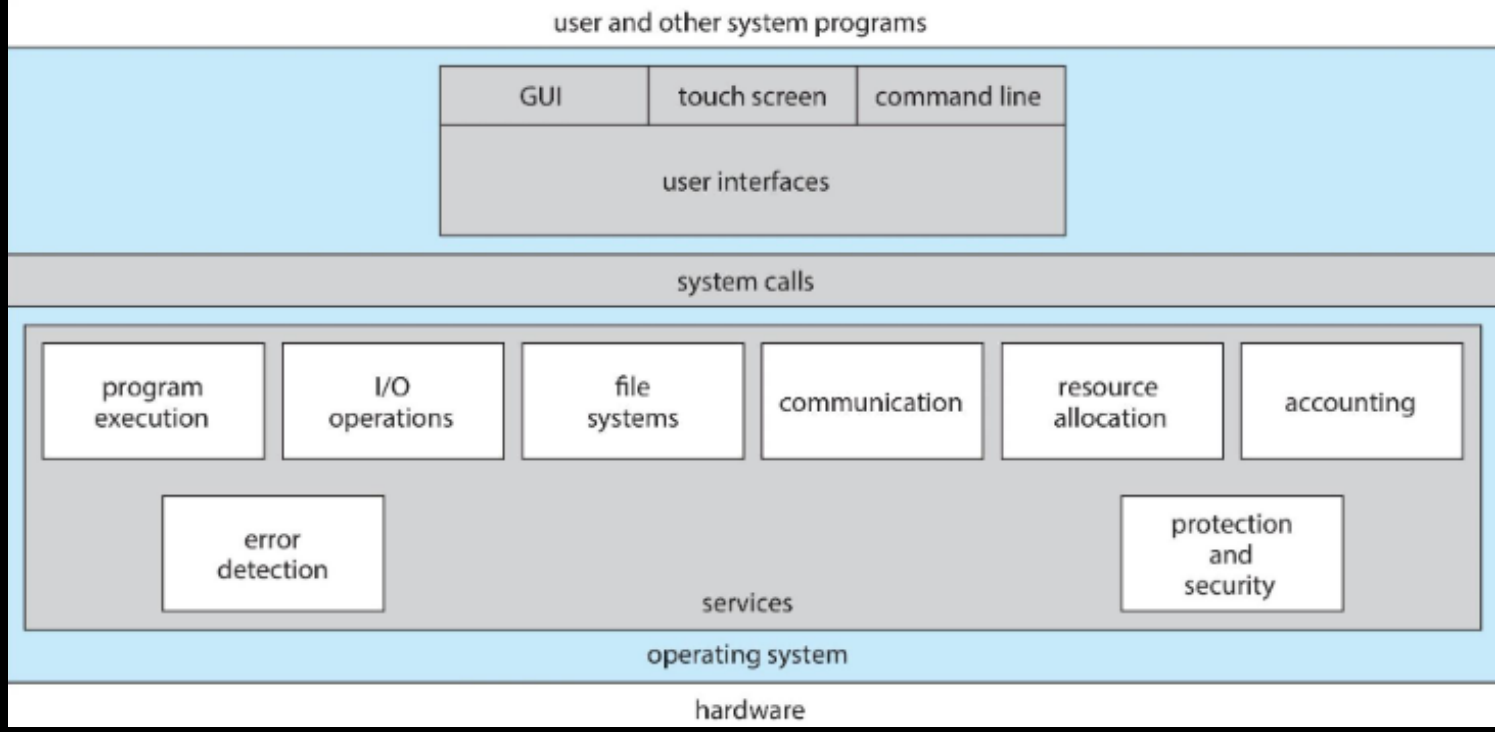




OS Structure

Services, structure, kernel, booting, user-kernel mode, system call, system program

OS Services



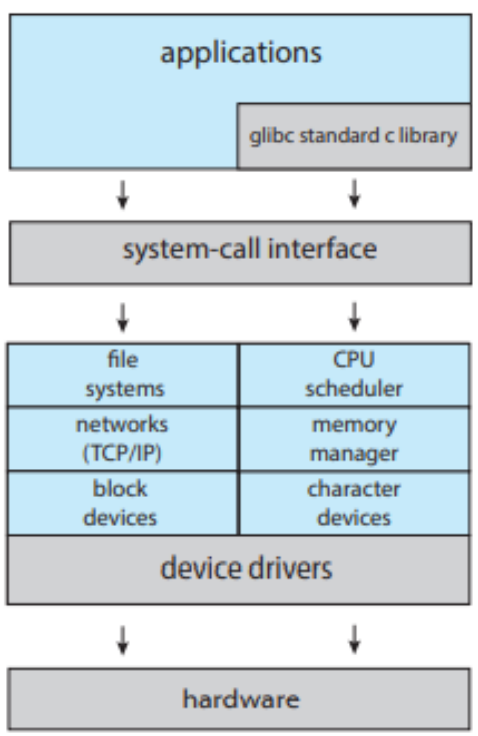
Ada beberapa service esensial di OS:

1. User Interface
   Bisa GUI, CLI, atau touch-screen.
2. Program Execution
   Load program ke memori, run/suspend/halt program, handle/display error.
3. I/O operations
   Interaksi antar perangkat I/O seperti disk, network. Biasanya diproteksi, jadi harus lewat syscall.
4. Filesystem manipulation
   Read/write/traverse filesystem, r/w files, enforcing permission, etc.
5. Communications
   Sharing informasi antar proses. Bisa melalui shared memory atau message passing.
6. Error Detection
   Deteksi error di CPU, memori, perangkat I/O, proses, dkk. Bisa recover dari kesalahan juga (halt, terminate, etc).

Beberapa juga tersedia tetapi untuk efisiensi system, bukan untuk membantu user:

1. Resource Allocation (pengalokasian resource ke beberapa proses)
2. Logging (tracking resource yang dipakai sebagai usage statistics)
3. Protection & Security (contohnya TPM, secure boot, dkk)

OS Structure



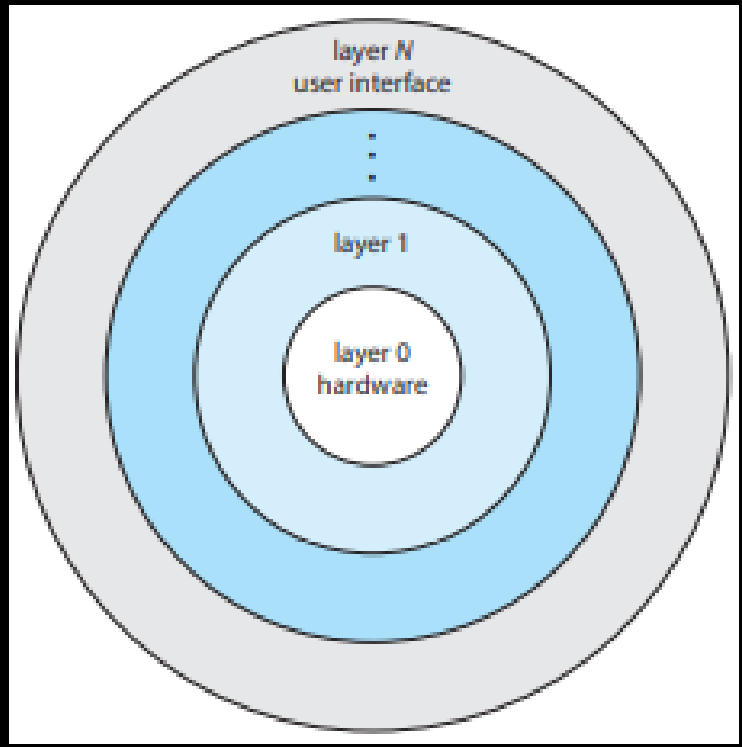
Monolithic

Ga punya struktur sama sekali, semua fungsi diletakin di kernel yang ada di satu file binary.

+ Performa bagus karena gaada overhead pemanggilan system call.

- Susah diextend karena tightly coupled, satu diubah semua bakal kena risk untuk berubah.

Contoh: MS DOS, Unix pre-Mach.



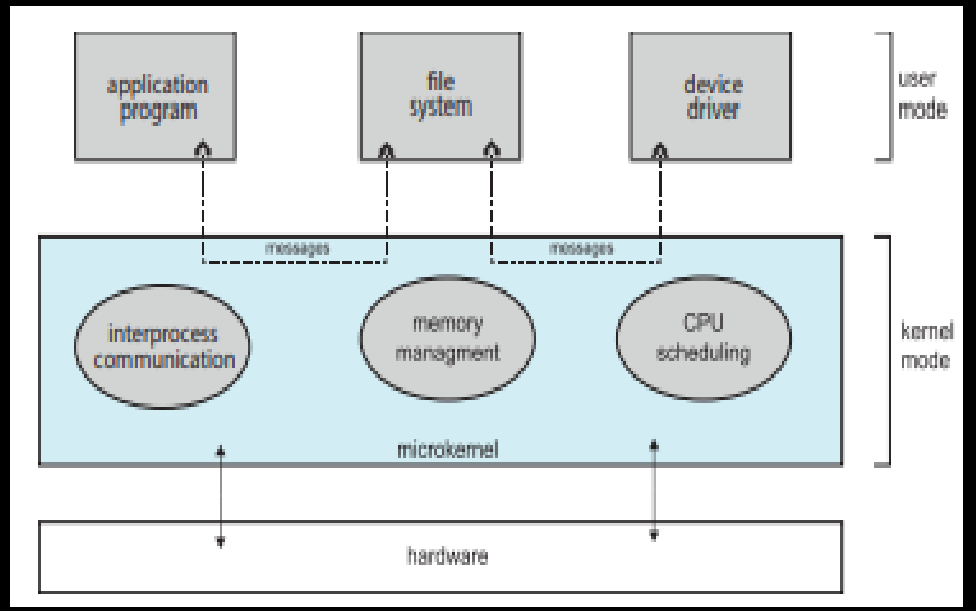
Layered

Dibagi menjadi N layer: layer 0 hardware, layer 1 kernel, layer N user interface, dengan tiap layer cuma bisa panggil/pake service dibawahnya doang.

+ Mudah didebug karena loosely coupled, menyimpelkan desain dan implementasi OS.

- Susah buat menentukan fungsi yang tepat untuk diletakkan di setiap layer. Performa juga ga bagus karena harus traverse tiap layer.

Contoh: Windows NT



Microkernel

Memisahkan komponen yang tidak esensial dari kernel menjadi user-level program.

+ Mudah untuk diextend, ramah memori, mudah di port ke arsitektur baru, lebih secure dan reliable karena jika ada yang salah, ga mengganggu kernel.

- Performa paling jelek, karena komunikasi antar mode mengharuskan data diduplikasi, belum lagi untuk ganti antar prosesnya.

Contoh: Mach (newest Unix), Darwin (based on Mach)

Kernel & Booting

Kernel merupakan core program yang berjalan pada sistem operasi.

Booting merupakan proses dimulainya computer dengan loading kernel. Proses booting biasanya sebagai berikut:

1. Kode kecil yang disebut bootstrap program atau bootloader mencari si kernel, kemudian dicopy ke memori dan dijalankan
2. Kernel menginisialisasi hardware
3. Root filesystem di mount.

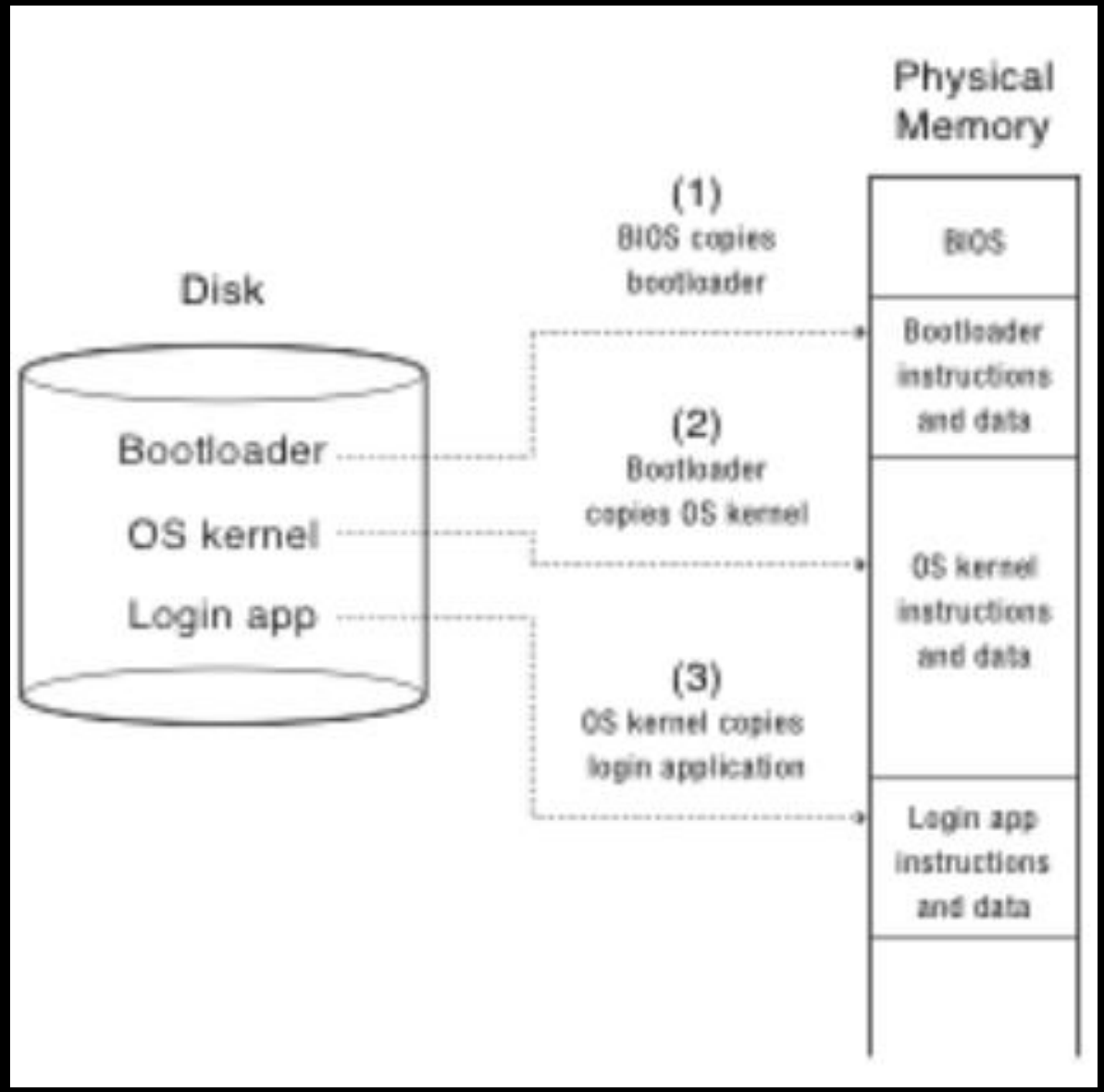
Ada dua jenis boot process:

1. Multistage boot process/ BIOS-based

Bootloader pertama, BIOS, ngeload bootloader kedua yang ada di boot block.

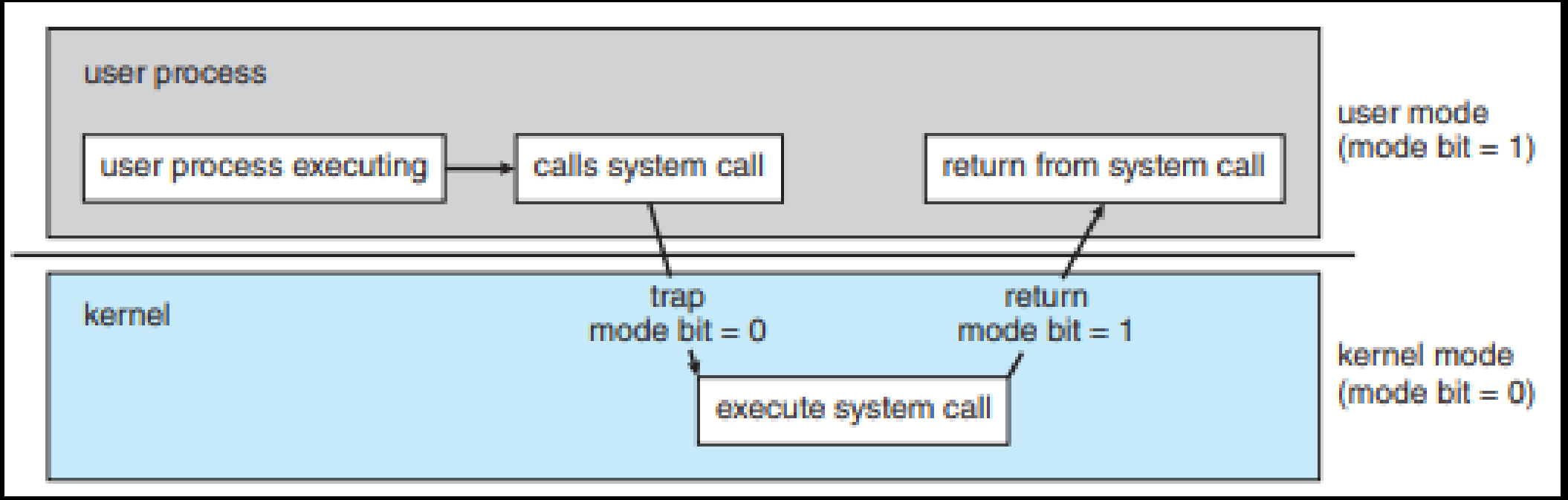
2. UEFI (Unified Extensible Firmware Interface)

Sama dengan BIOS, tapi nyimpen konfigurasinya di partisi special dengan format .efi daripada di CMOS. Punya banyak kelebihan dibanding BIOS.



User / Kernel Mode

- Instruksi program dapat dijalankan dengan user mode atau kernel mode. Hal ini dilakukan supaya system operasi dapat memproteksi dirinya sendiri dan komponen system lainnya dari malicious atau incorrect program.
- Biasanya user punya mode bit 1, sedangkan kernel 0.
- Dalam x86, mode disimpan di EFLAGS, sedangkan MIPS di status register
- Pada kernel mode, didapatkan full privileges tanpa batasan tertentu. Di user mode, terdapat banyak batasan dan hanya beberapa perintah yang diizinkan system operasi.
- Untuk CPU yang memiliki support hardware (dual-mode), privileged instruction bakal hanya bisa dijalankan di kernel, kemudian user mode bakal dilimit akses memori biar ga overwrite kernel, terakhir ada timer untuk membatasi lama berjalannya proses di kernel mode



System Call & API

- System call memberikan sebuah interface ke service yang ada di system operasi. Biasanya dalam bentuk fungsi di C atau C++ yang dibungkus dalam API.
- Application Program Interface (API) memberikan serangkaian set fungsi yang tersedia kepada pengembang software, termasuk parameter yang di pass ke fungsinya dan mengembalikan nilai yang bisa diekspektasi oleh programmer.
- API menyembunyikan implementasi sebenarnya dari system call. Misalnya di C ada printf yang sebenarnya manggil write() system call. Jadi, programmer cukup mengikuti apa yang ada di API tanpa tau implementasi aslinya gimana.
- 3 metode parameter passing: basically orkom (review calling convention) + parameter passing via table.
- Tipe system call: process control (create/terminate/load/execute process), file management (create/delete/open/close file), device management (request/release/read/write devices), information maintenance (time or date, attributes info), communications (create/delete connection, send/receive message), protection (file permission)

System Programs

- Sistem program (juga bisa disebut sebagai system service) memberikan environment yang nyaman untuk pengembangan dan eksekusi software. Bisa dibilang sebagai user interface ke system calls.
- Beberapa kategori dan contoh system programs:

a. File Management (File Explorer, cp/rm/ls/mv dst)

b. State Information (Registry Editor, Event Viewer, top, Task Manager, dst)

c. File Modification (Notepad, dd, nano, dst)

d. Programming-language support (GCC, Clang, MSVC, dst)

e. Program loading and execution (gdb, dst)

f. Communications (ssh, scp, wget, curl, ping, dst)

g. Background services (daemon, windows services, dst)

- Ada juga application program yang bukan bagian dari OS (at least yang gak di ship secara default di OSnya), misalnya Word, PowerPoint (ini!), games, dst.

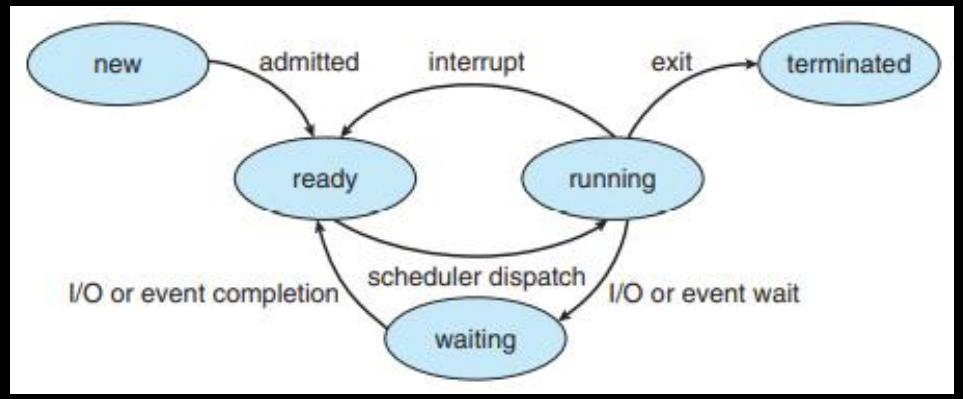
Process

Concept, scheduling, operations (creation, termination), co-op, IPC

Process Concept

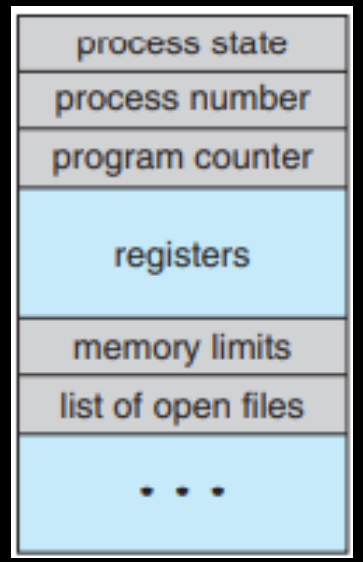
- Process = program yang dieksekusi dengan progress yang sekuensial. Terdiri dari program counter, text, data, heap dan stack (review orkom: memory layout).
- Sebuah proses memiliki 5 state:

1. New = Proses sedang dibuat.
2. Running = Instruksi program sedang dieksekusi.
3. Waiting = Proses menunggu sebuah event muncul.
4. Ready = Proses menunggu untuk ditugaskan ke prosesor.
5. Terminated = Proses telah selesai dieksekusi.



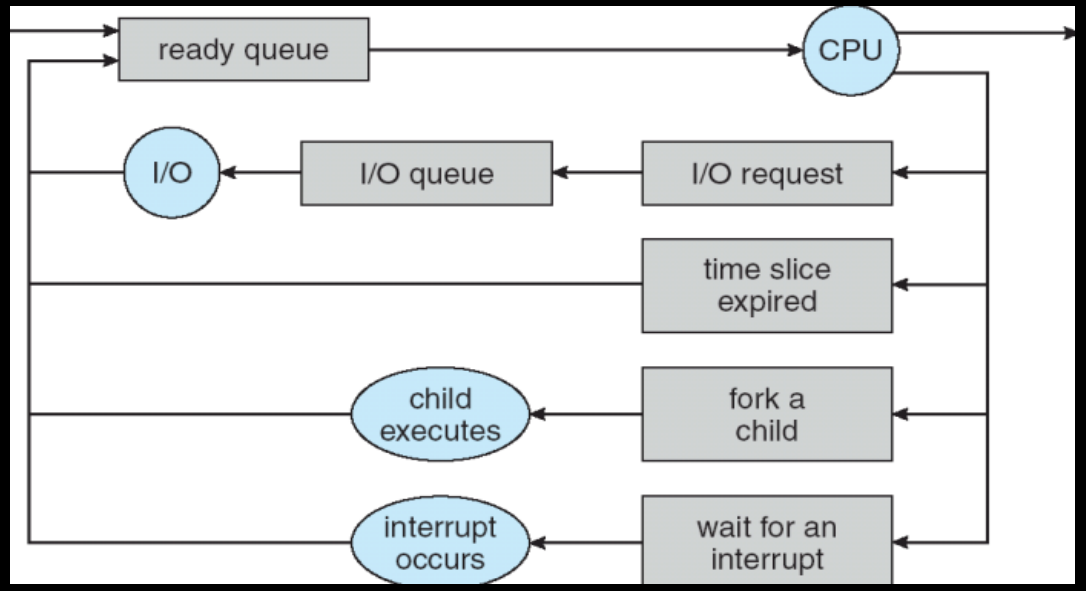
- Process Control Block (PCB) merupakan representasi proses dalam OS. Terdiri dari:

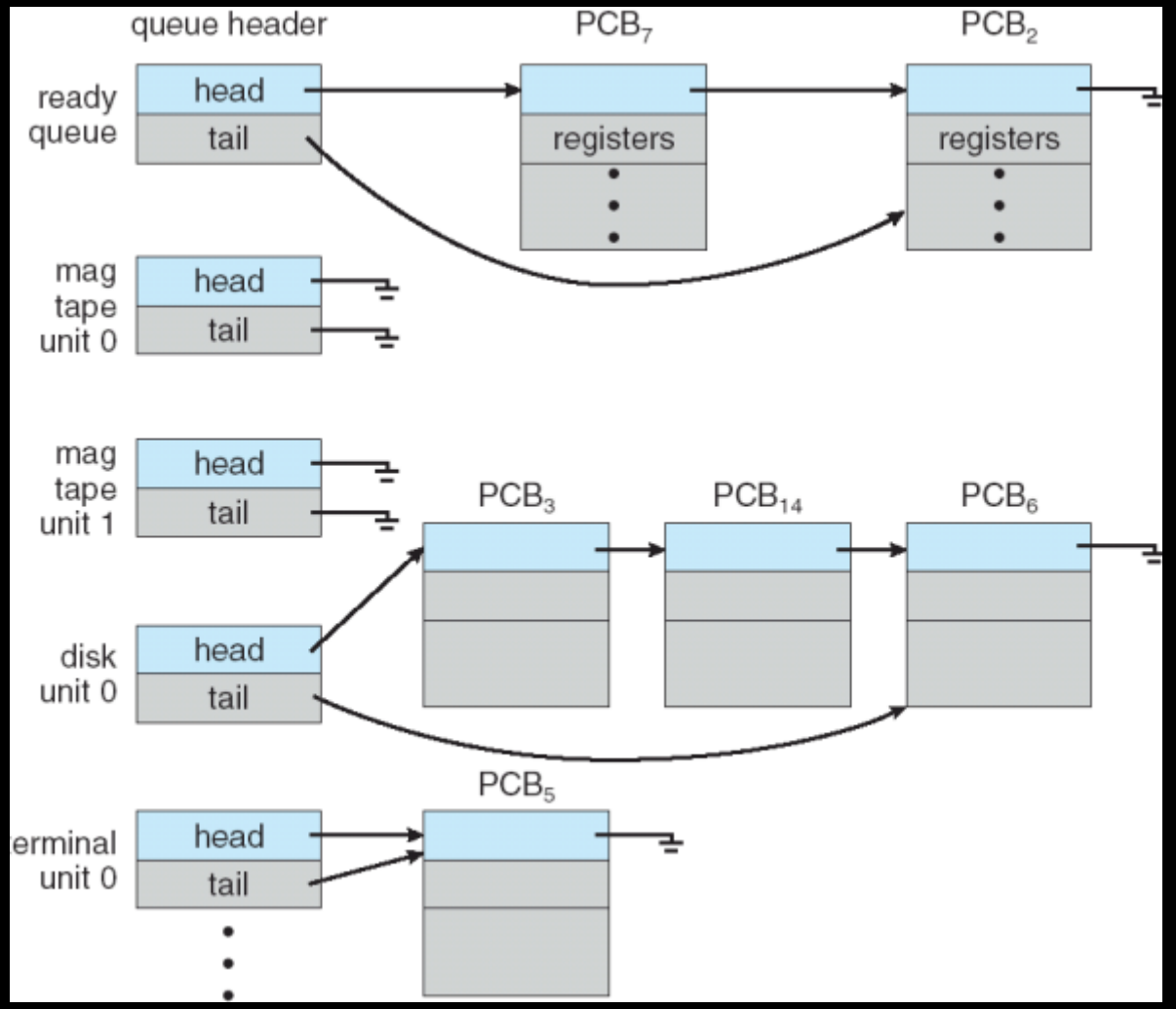
1. Process state: state dari process
2. Program counter: address dari instruksi selanjutnya untuk dieksekusi
3. CPU registers: nilai register saat itu
4. CPU-scheduling info: priority, scheduling params
5. Memory-management info: page, segment table
6. Accounting info: nomor proses, time limit
7. I/O status: file yang dibuka, device yang dialokasikan



Process Scheduling

- Process scheduler = mengambil proses yang tersedia untuk dieksekusi ke CPU core.
- Ada beberapa jenis queue untuk process scheduling (dalam bentuk linked list):
- Job queue = semua proses dalam system
- Ready queue = semua proses di main memory, ready dan waiting untuk dieksekusi
- Device queue = semua proses yang menunggu I/O device
- Proses dapat berpindah-pindah pada scheduling queue.

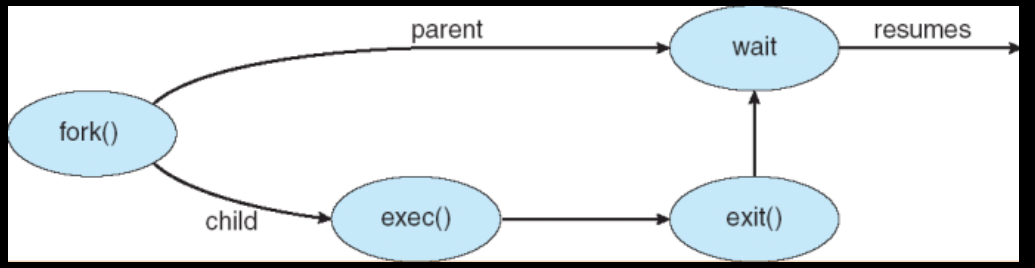




Process Scheduling

- Ada dua jenis scheduling (+ 1 opsional)
  a. Long term scheduler / job scheduler:
     - Mengambil proses yang perlu dibawa ke ready queue.
     - Dari queue yang lain ke ready queue.
     - Pemanggilannya sangat jarang (dalam detik/menit), jadi mungkin lama.
     - Mengontrol degree of multiprogramming (banyaknya core CPU di CPU multicore)
  b. Short term scheduler / CPU scheduler
     - Mengambil proses mana yang dieksekusi selanjutnya dan mengalokasikan CPU.
     - Dari ready queue ke CPU.
     - Pemanggilannya sangat sering (dalam milisekon), jadi harus cepat.
  c. Medium term scheduler: Ada mekanisme untuk swap out proses saat di CPU.
- Process dapat bounded dengan:
  - I/O bound: lebih sering melakukan I/O. Banyak CPU burst pendek. Contohnya word.
  - CPU bound: lebih sering melakukan komputasi. Sedikit CPU burst lama. Contohnya training ML.
- Ketika context switching, system harus save state proses lama dan load saved state proses baru. Biasanya dianggap overhead dan waktunya dependen sama hardware.

Process Operation



Parent Creation

- Parent process membuat child process yang membuat proses lain, membentuk tree dari proses.
- Resource: parent dan child bisa antara share semua resource, child share subset dari resource parent, atau parent dan child ngga share resource sama sekali.
- Execution: parent bisa berjalan secara konkuren dengan child atau parent bisa menunggu child untuk terminate.
- Address space: child menduplikasi parent, atau child berupa program yang diload ke child process yang dibuat oleh parent.
- Dalam unix, fork membuat proses baru, sedangkan exec mengubah memory space proses setelah fork dengan program baru.

```
int main()
{
Pid_t  pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
            fprintf(stderr, "Fork Failed");
            exit(-1);
    }
    else if (pid == 0) { /* child process */
            execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
            /* parent will wait for the child to complete */
            wait (NULL);
            printf ("Child Complete");
            exit(0);
    }
}
```

Parent Termination

Proses mengeksekusi instruksi terakhir dan melakukan exit (meminta system operasi untuk menghapusnya). Parent dapat mengoutput data child dengan wait dan resource proses akan didealokasi oleh OS.
Parent bisa saja terminate eksekusi child (abort), misalnya jika child udah exceeded allocated resources. Task yang diassign ke child jadi ga perlu. Kalau parent exiting, beberapa OS biasanya bakal terminate semua childnya (cascading termination)

Coop (Cooperating Process)

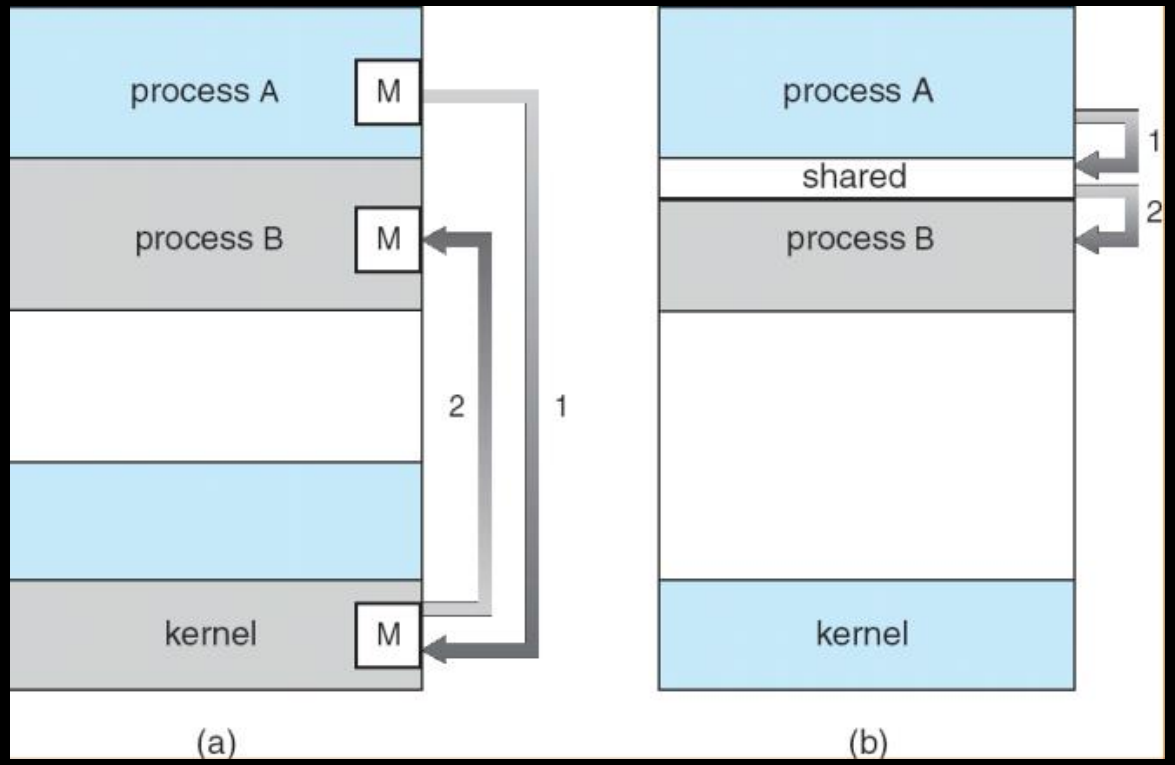
- Proses independent tidak dapat mempengaruhi atau dipengaruhi oleh proses yang lain.
- Proses cooperating dapat mempengaruhi atau dipengaruhi oleh proses yang lain.
- Manfaat coop: ya kaya main genshin atau game coop

1. Sharing info (ngasih tau lokasi musuh)
2. Computation speed-up (dibantu nyelesain puzzle genshin)
3. Modularity (tiap player bisa punya role berbeda: tanker, healer, dps? gitu2 lah)
4. Convenience (siapa sih yang ga mau cepet nyelesain domain??)

- Komunikasi antara proses yang berkooperasi membutuhkan komunikasi yang disebut Interprocess Communication (IPC)

Interprocess Communication (IPC)

- IPC = komunikasi antara proses yang berkooperasi sehingga dapat melakukan sinkronisasi data / aksi.
- Ada dua model fundamental untuk IPC:
    1. Shared Memory
       Producer-consumer problem: producer proses menghasilkan informasi (diisiin) yang kemudian dikonsumsi (dikosongin) oleh consumer process. Kedua proses harus saling tersinkronisasi sehingga tidak ada consumer yang mencoba untuk mengonsumsi item yang belum diproduksi.
    2. Message Passing



Message Passing

- Terdapat dua operasi: send(message) dan receive(message).
- Untuk berkomunikasi, kedua proses harus melakukan communication link dan exchange pesan dengan send/receive. Communication link bisa berupa physical maupun logical, dan implementasinya dapat dibedakan menjadi dua:

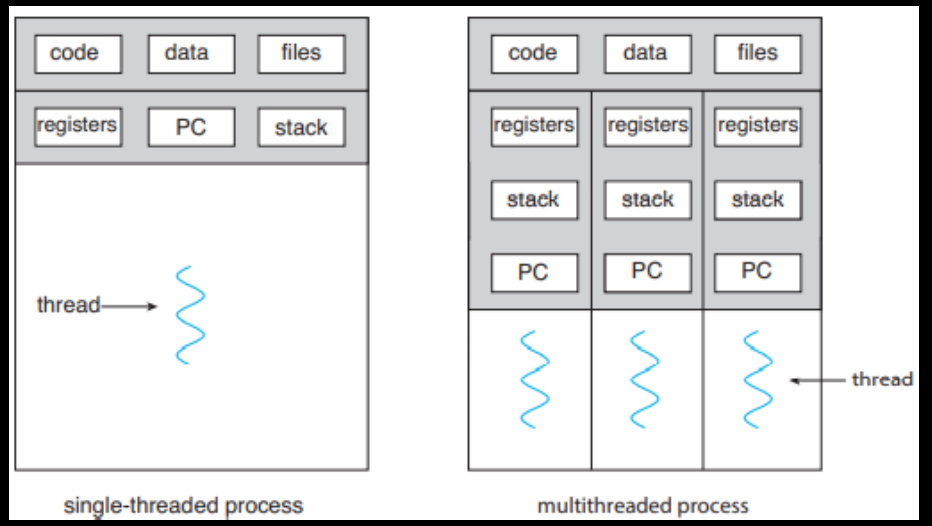
1. Direct Communication
   Proses harus secara eksplisit menyebutkan namanya satu sama lain: send(P, m) dan receive(Q, m). Link komunikasi akan terbuat secara otomatis dan sebuah link diasosiasikan dengan satu pair proses yang berkomunikasi. Untuk setiap pair, terdapat hanya satu link, dan link biasanya bersifat bi-directional, meskipun bisa unidirectional.
2. Indirect Communication
   Terdapat mailbox yang punya id unik dan dapat menerima pesan: send(A, m) dan receive(A, m) dengan A = mailbox. Link komunikasi dibuat jika beberapa proses sharing mailbox yang sama dan dapat diasosiasikan dengan banyak proses. Tiap pair dari proses mungkin sharing beberapa link komunikasi berbeda dan link bersifat unidirectional maupun bi-directional.
   Masalah terjadi jika P1,P2,P3 share mailbox A dengan P1 pengirim dan P2, P3 penerima. Untuk menyelesaikan persoalan, implementasi dapat diubah dengan salah satu solusi berikut: link harus diasosiasikan maksimal dua proses, hanya diperbolehkan satu proses untuk menerima pesan, atau system memilih penerima secara acak (pengirim akan diberitahu siapa yang menerima pesannya).

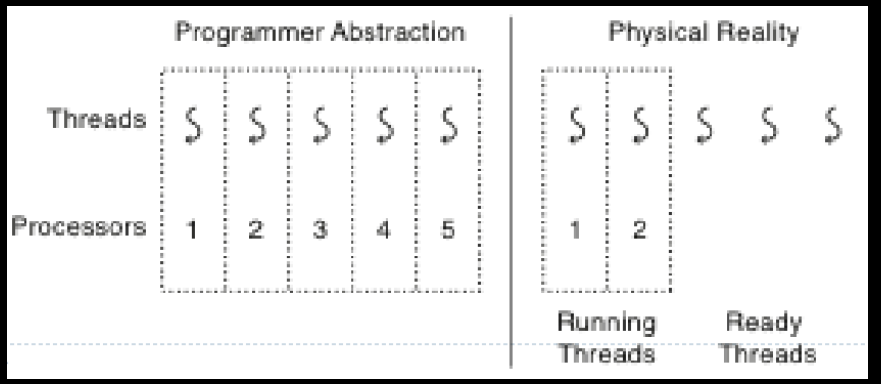
Threads

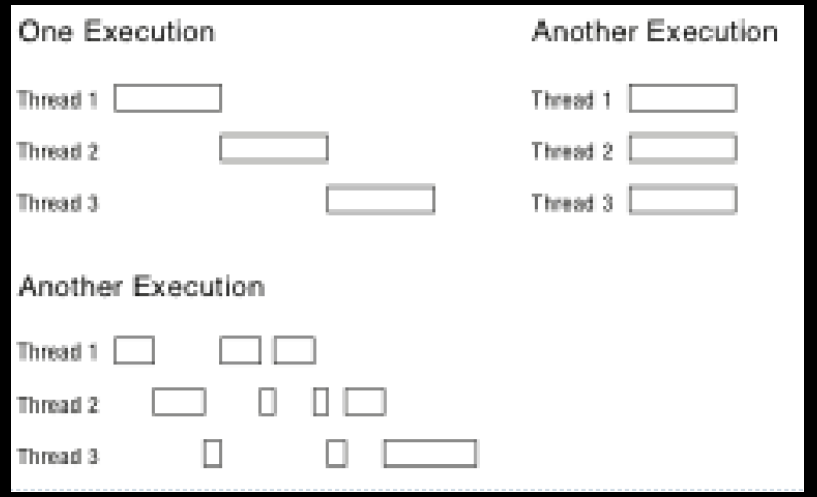
Definisi, TCB, kernel vs user, operations, models, issues

Definisi

- Threads = sekuens eksekusi tunggal (model pemrograman sederhana) yang merepresentasikan task yang dapat dijadwalkan tersendiri (OS dapat menjalankan atau mensuspend thread kapan saja).
- Single-thread = Hanya bisa memproses satu thread saja. Multi-thread = Dapat memproses lebih dari satu thread
- Concurrency = Beberapa thread yang membuat progress. Parallelism = Beberapa thread yang membuat progress secara bersamaan.
  Adalah mungkin untuk mendapatkan concurrency tanpa parallelism. Dalam single-core CPU, hanya mungkin terjadi concurrency (CPU scheduler membuat ilusi parallelism dengan berganti proses dalam waktu yang sangat singkat), sedangkan multicore CPU, bisa terjadi parallelism.
  Oleh karena itu, eksekusi program harus dirancang untuk dapat dilakukan dengan berbagai macam penjadwalan.
- Benefit thread: responsiveness (aplikasi interaktif dapat terus berjalan saat melakukan operasi yang sangat lama), resource sharing (thread secara default sharing memory dan resource), economy (membuat proses baru harus melakukan copy memory, sedangkan thread cukup sharing memory. Context-switching thread juga lebih cepat daripada process), scalability (dapat menggunakan full capability dari perangkat manapun).





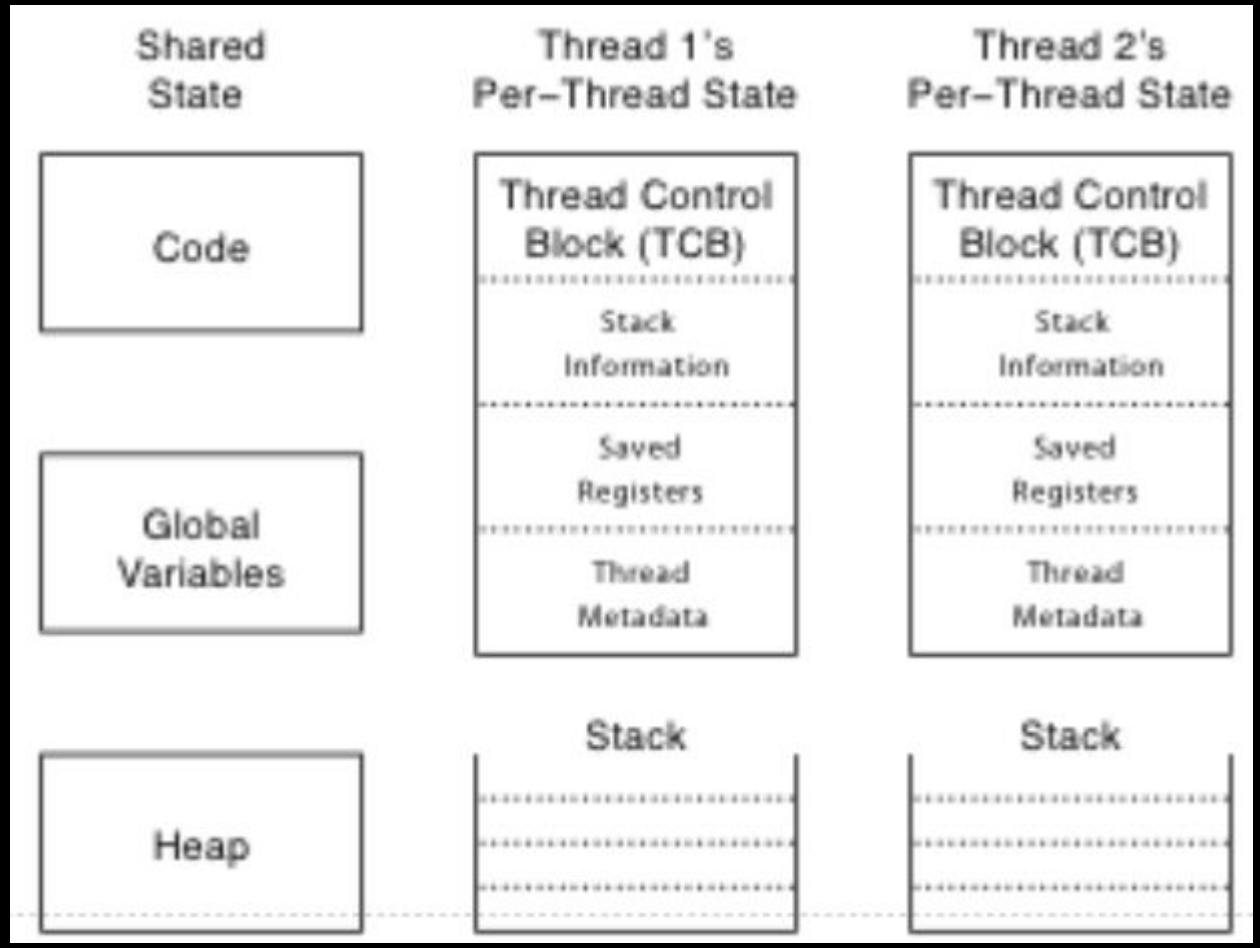


Thread Control Block

- Thread juga memiliki control blocknya sendiri layaknya process, tetapi ia mempunyai beberapa state yang di-share.
- Beberapa state yang dishare sesama thread:

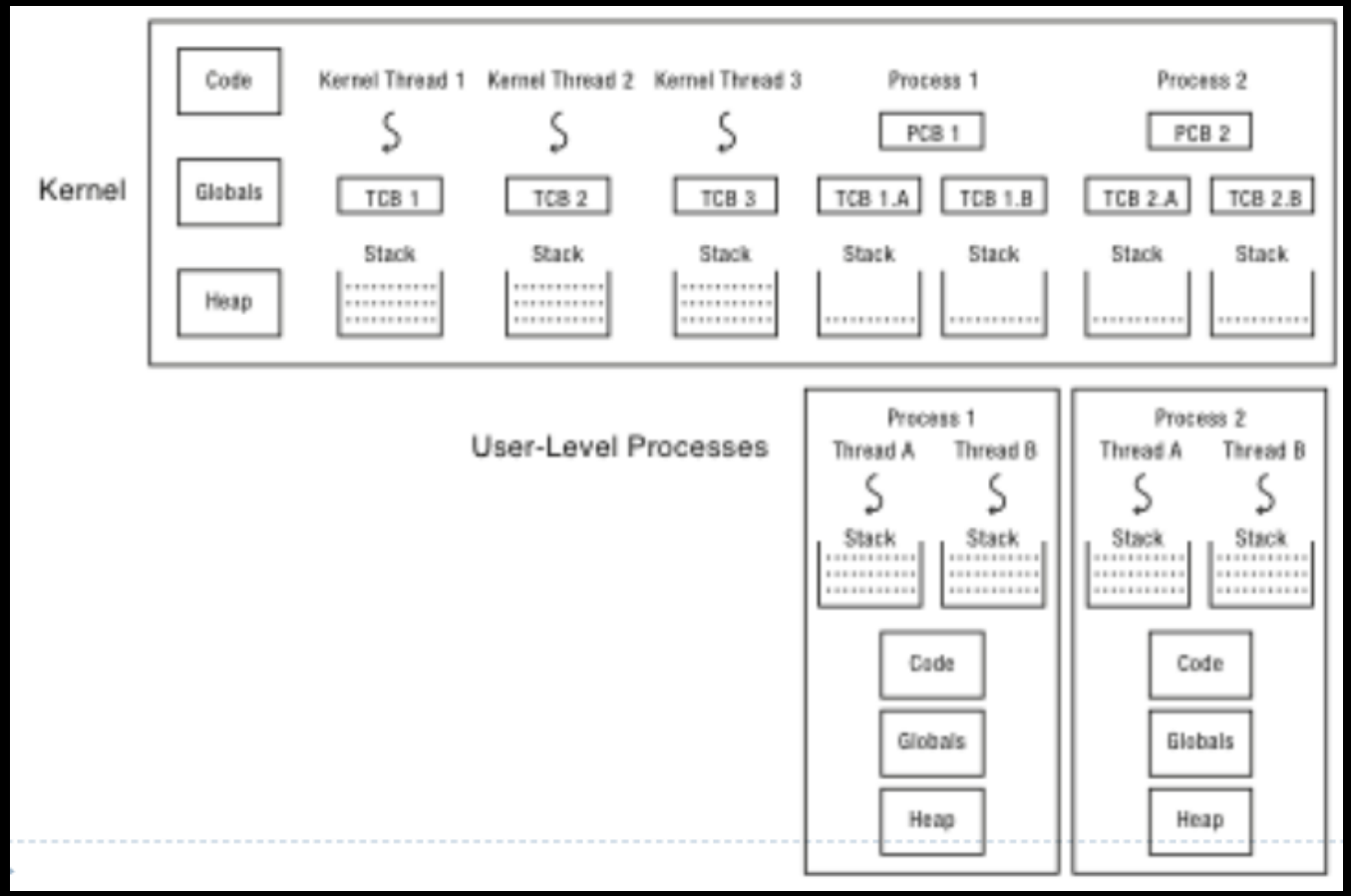
- Code
- Global Variables
- Heap

- Stack, registers, dan metadata akan berbeda setiap thread.



Kernel vs User Thread

- Untuk kernel thread, abstraksi threadnya cuma bisa di kernel. Operasi di kernel thread cuma hanya bisa dilakukan via system call.
- User thread operasinya bisa dilakukan sendiri melalui library (pthreads di POSIX, Win32 threads, atau Java threads) tanpa system call. Bagi si kernel, ia gak tau tentang user thread dan nganggep mereka sebagai single threaded user process
- Contoh disamping merupakan multithreaded user process dengan kernel thread.



Thread Operations

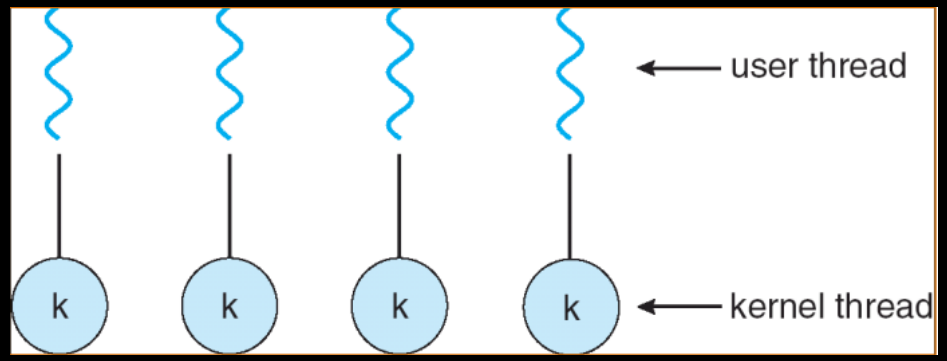
- thread_create(thread, func, args)
  Membuat thread baru yang mengeksekusi func(args).
- thread_yield()
  Melepaskan processor yang digunakan (untuk digunakan thread lainnya) dan direschedule.
- thread_join(thread)
  Menunggu thread selesai, kemudian mendapatkan return valuenya.
- thread_exit(retval)
  Terminate thread dan membangkitkan thread join dengan return value si retval.

```
#define NTHREADS 10
thread_t threads[NTHREADS];
main() {
   for (i = 0; i < NTHREADS; i++)  thread_create(&threads[i], &go, i);
   for (i = 0; i < NTHREADS; i++) {
      exitValue = thread_join(threads[i]);
      printf("Thread %d returned with %ld\n", i, exitValue);
   }
   printf("Main thread done.\n");
}
void go (int n) {
   printf("Hello from thread %d\n", n);
   thread_exit(100 + n);
   // REACHED?
}
```

```
bash-3.2$ ./threadHello
Hello from thread 0
Hello from thread 1
Thread 0 returned 100
Hello from thread 3
Hello from thread 4
Thread 1 returned 101
Hello from thread 5
Hello from thread 2
Hello from thread 6
Hello from thread 8
Hello from thread 7
Hello from thread 9
Thread 2 returned 102
Thread 3 returned 103
Thread 4 returned 104
Thread 5 returned 105
Thread 6 returned 106
Thread 7 returned 107
Thread 8 returned 108
Thread 9 returned 109
Main thread done.
```

Thread Model

Ketika user thread memanggil system call, bagaimana caranya untuk mapping ke kernel thread?



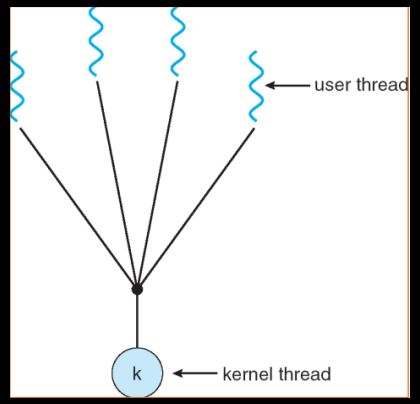
One to One

Satu user thread dimapping ke satu kernel thread

\+ Gak perlu khawatir kalau ada user thread yang blocking.

\- Banyak kernel thread yang harus dibuat, which is expensive to create.

Contoh: Windows NT/XP/2000, Linux, Solaris 9 dan diatasnya



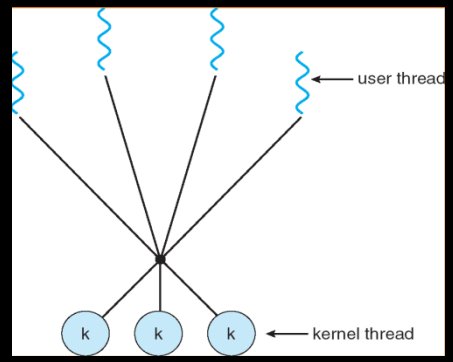
Many to One

Beberapa user thread dimapping ke satu kernel thread

\+ Lebih dikit kernel thread.

\- Kalau ada user thread yang ngeblock, semuanya (dlm kernel thread yg sama) ikut keblock.

Contoh: GNU Portable Thread



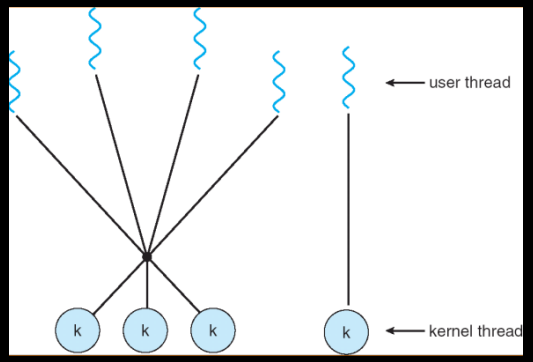
Many to Many

Beberapa user thread dimapping ke beberapa kernel thread.

\+ Concurrency achieved!

\- Hati-hati kebanyakan kernel thread.

Contoh: Windows NT/2000 + ThreadFiber, Solaris diatas versi 9



Two-level

Sama seperti many-to-many, tapi satu user thread bisa dibound ke satu kernel thread.

\+ atau – kurleb sama dgn M:M

Contoh: Irix, Solaris 8 dan versi sebelumnya

Thread Issues

- Semantik fork() dan exec(): apakah fork() menduplikasi thread yang memanggil, atau seluruh threadnya?
- Thread cancellation: bisakah si thread diberhentikan sebelum selesai? Solusinya ada 3, asynchronous cancellation (secara langsung memberhentikan thread), deferred cancellation (check target thread apakah harus untuk diberhentikan secara periodik), atau not cancellable (ga bisa diberhentikan sama sekali).
- Signal Handling: di UNIX, untuk menotifikasi proses bahwa sebuah event muncul, digunakan signal. Bagaimana caranya untuk mengirimkan signal ke thread? Solusinya ada 4, thread which applies (thread yang bisa menerima signal), all thread (semua thread dalam process), certain thread (beberapa thread dalam process), atau one thread signal handler (satu thread yang menerima semua signal).
- Thread pools: membuat beberapa thread di dalam pool sebagai tempat menunggu mereka bekerja. Alasannya karena pake thread yang udah dibuat lebih cepat daripada bikin thread baru.
- Thread specific data atau thread local storage: memberikan thread berupa copy dari data dirinya. Berguna kalau kita ga punya control terhadap thread creation (misalnya dalam menggunakan thread pool). Mirip seperti variable statis yang terpisah di setiap thread.
- Scheduler activations: model Many to Many atau Two Level membutuhkan suatu mekanisme komunikasi untuk mengubah banyaknya kernel thread yang ada di proses secara dinamis (untuk best performance). Scheduler activation ini memberikan mekanisme komunikasi dari kernel ke library thread melalui upcalls, yang memberi tahu jika thread akan terblock atau thread dapat berjalan Kembali.

Scheduling

Concept, criteria, algorithms [FCFS, SJF, Priority, RR, Multilevel Queue, Multilevel Feedback], thread scheduling, multiprocessing, real-time [Priority, Rate-Monotonic, EDF]

Concept

- Dalam konsep multiprogramming, kita ingin untuk menggunakan CPU semaksimal mungkin jadi ngga nganggur.
- Selama eksekusi proses, biasanya akan terlibat siklus eksekusi CPU dan I/O wait (CPU burst dan I/O burst) yang saling bergantian.
- CPU Scheduler = memilih proses di memori yang siap untuk dieksekusi dan mengalokasikan CPU ke salah satu proses tersebut.
- CPU scheduler dapat terjadi ketika proses dalam keadaan running menjadi waiting (non-preemptive) atau ready (preemp), dari waiting menjadi ready (preemp), dan termination (non-preemptive).
- Dispatcher = modul yang memberikan control CPU ke proses yang terpilih oleh short-term scheduler (CPU scheduler). Modul ini melibatkan context switching, switching ke user mode, dan melompat ke bagian program yang terhenti sebelumnya untuk dilanjutkan.
- Dispatch latency = waktu yang dibutuhkan dispatcher untuk memberhentikan proses dan melanjutkan proses lain

Criteria

- [Max] CPU Utilization = membuat CPU sesibuk mungkin.
- [Max] Throughput = banyaknya proses yang selesai dieksekusi per satuan waktu
- [Min] Turnaround time = waktu yang dibutuhkan untuk mengeksekusi proses
- [Min] Waiting time = waktu yang dibutuhkan proses untuk menunggu di ready queue
- [Min] Response time = waktu yang dibutuhkan sejak request disubmit hingga response pertama dibuat, bukan output (biasanya ada pada lingkungan time-sharing)

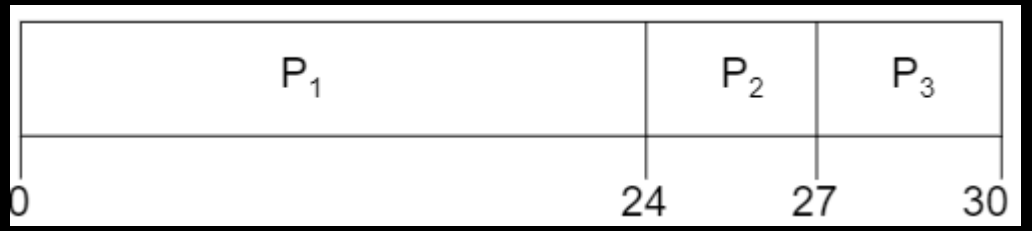
Konsep tambahan:

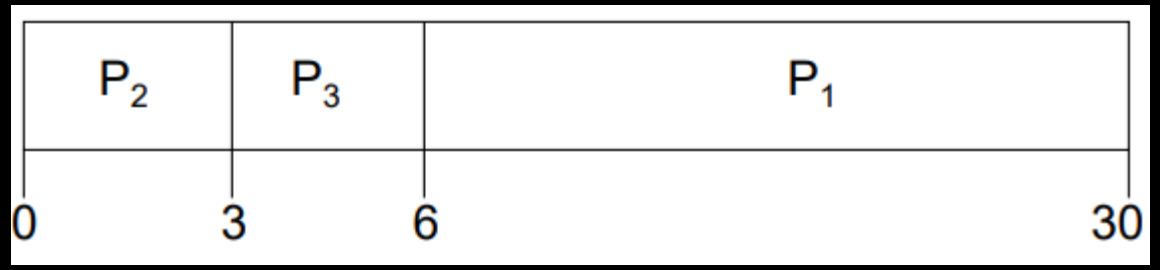
- Preemptive = Keadaan yang memungkinkan proses yang baru datang untuk menggantikan proses yang sedang dieksekusi secara paksa (proses yang sedang dieksekusi akan Kembali diletakkan pada ready queue dengan sisa waktunya).
- Nonpreemptive = Keadaan yang mengharuskan semua proses yang baru datang untuk menunggu CPU jika sedang mengeksekusi proses yang lain.
- Gantt chart = grafik penjadwalan sebuah aktivitas/suatu hal terhadap waktu.
- CPU Efficiency = CPU time yang tidak idle dibagi dengan total CPU time

First Come-First Served (FCFS)

Process	Burst Time
P_1	24
P_2	3
P_3	3

Jika diketahui yang datang adalah P1, P2, dan P3 secara berurutan, maka
Waiting time untuk P1 = 0; P2 = 24; P3 = 27. Averagenya (0 + 24 + 27)/3 = 17





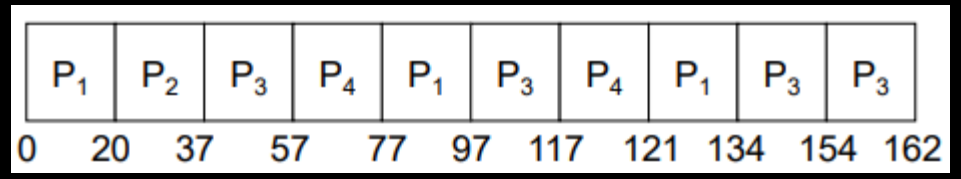
Jika diketahui yang datang adalah P2, P3, dan P1 secara berurutan, maka
Waiting time untuk P1 = 6; P2 = 0; P3 = 3.
Averagenya (6 + 0 + 3)/3 = 3.
Jauh lebih singkat daripada yang sebelumnya.

- Algoritma FCFS ini bersifat nonpreemptive, sehingga proses yang lain tidak boleh menyerobot proses yang sedang berjalan, mengakibatkan convoy effect.
- Convoy effect = Fenomena dimana beberapa thread menunggu sebuah thread yang sangat lama dieksekusi (layaknya konvoi) hingga selesai. Hal ini mengakibatkan penggunaan CPU dan device yang tidak maksimal.

Round Robin (RR)

- Round robin scheduling sebenarnya sama dengan FCFS, hanya saja ditambahkan preemptive jika waktu eksekusi melebihi suatu unit waktu (biasanya disebut time quantum atau time slice) yang secara umum berkisar antara 10-100ms.
- Dalam RR scheduling, ready queue tetap bersifat FIFO, tetapi dianggap sebagai queue yang sirkuler.
- Jika Q = time quantum, T = burst time, maka dibutuhkan setidaknya $\left\lceil \frac{T}{Q} \right\rceil - 1$ kali context switch untuk proses tersebut. Karena context switching memerlukan waktu yang sejatinya adalah overhead, maka Q harus dibuat sedemikian mungkin sehingga tidak terlalu kecil (banyak overhead untuk context switch) maupun terlalu besar (kembali ke bentuk FCFS).

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24



- Biasanya RR mempunyai rata-rata turnaround yang lebih tinggi daripada SJF, tetapi memiliki response yang lebih baik.

Priority Scheduling

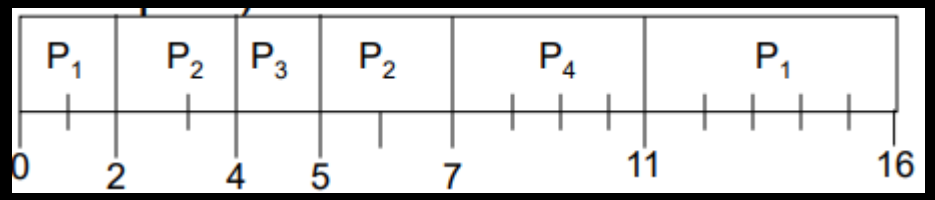
- Jika terdapat angka prioritas untuk setiap proses, CPU akan dialokasikan ke proses yang memiliki prioritas paling tinggi (angka paling kecil = prioritas paling tinggi).
- Permasalahan dari priority scheduling adalah starvation, dimana proses dengan prioritas paling kecil bisa saja tidak akan tereksekusi
- Solusi dari permasalahan tersebut adalah aging, dimana tiap proses dalam ready queue akan ditinggikan prioritasnya seiring waktu.
- Salah satu contoh CPU scheduling yang berbasis prioritas adalah SJF (shortest job first), dimana prioritasnya adalah prediksi CPU burst time selanjutnya.

Shortest Job First (SJF)

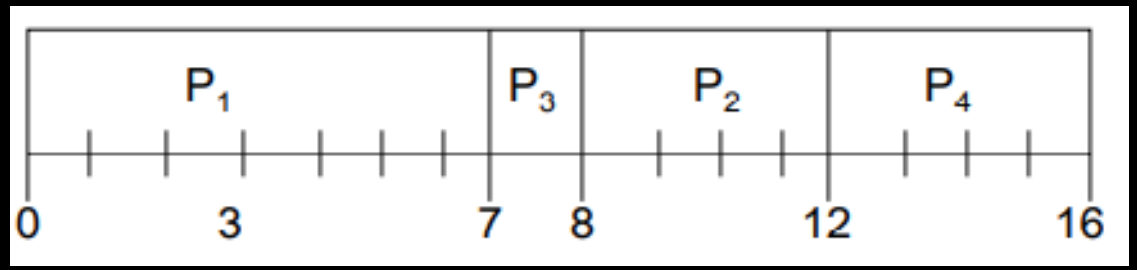
- Dengan mengasosiasikan setiap proses dengan panjang burst CPU selanjutnya, SJF menjadwalkan proses menurut panjang tersebut dimulai dari paling singkat hingga paling lama, sehingga menghasilkan penjadwalan yang optimal dengan rata-rata waiting time minimum.
- Dua skema yang dipakai:

1. nonpreemptive jika proses yang berjalan tidak bisa "diserobot" (preempted) hingga menyelesaikan CPU burstnya.
2. preemptive jika sebuah proses yang datang dapat menyerobot proses yang berjalan dengan syarat panjang CPU burstnya lebih kecil daripada waktu sisa eksekusi proses yang sedang berjalan. Biasanya disebut dengan shortest-remaining-time-first scheduling.

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4



Contoh kasus non-preemptive, gantt charnya:



Average waiting timenya = (0 + 6 + 3 + 7)/4 = 4

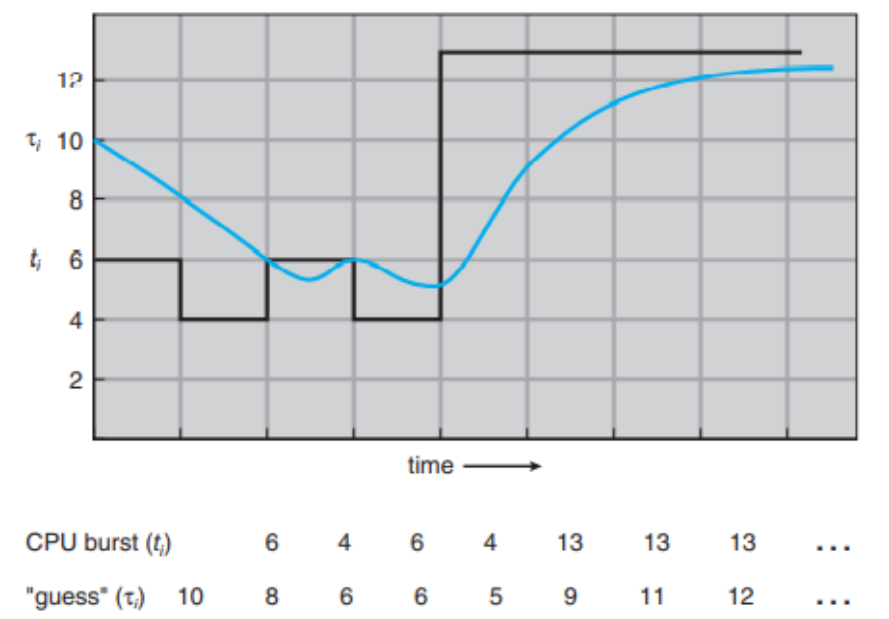
Untuk kasus preemptive, gantt chartnya diperoleh disamping
Average waiting timenya = (9 + 1 + 0 +2)/4 = 3

Shortest Job First (SJF) – Exponential Average

Meskipun penjadwalan SJF optimal, tetapi kita tidak bisa mengimplementasikannya dalam CPU scheduling karena gak ada caranya untuk tau CPU burst selanjutnya. Untuk dapat diimplementasikan, CPU burst selanjutnya dapat diprediksi dengan aproksimasi exponential average, dimana untuk semua $\alpha, 0 \leq \alpha \leq 1$:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

Dengan τ_{n+1} adalah prediksi CPU burst selanjutnya, τ_n adalah prediksi CPU burst sebelumnya, dan t_n adalah panjang CPU burst sekarang. Jika $\alpha = 0$, maka $\tau_{n+1} = \tau_n$ yang berarti CPU burst sekarang **tidak** diperhitungkan. Jika $\alpha = 1$, maka $\tau_{n+1} = t_n$ yang berarti CPU burst sekarang diperhitungkan.



Jika formula diperpanjang hingga τ_0,

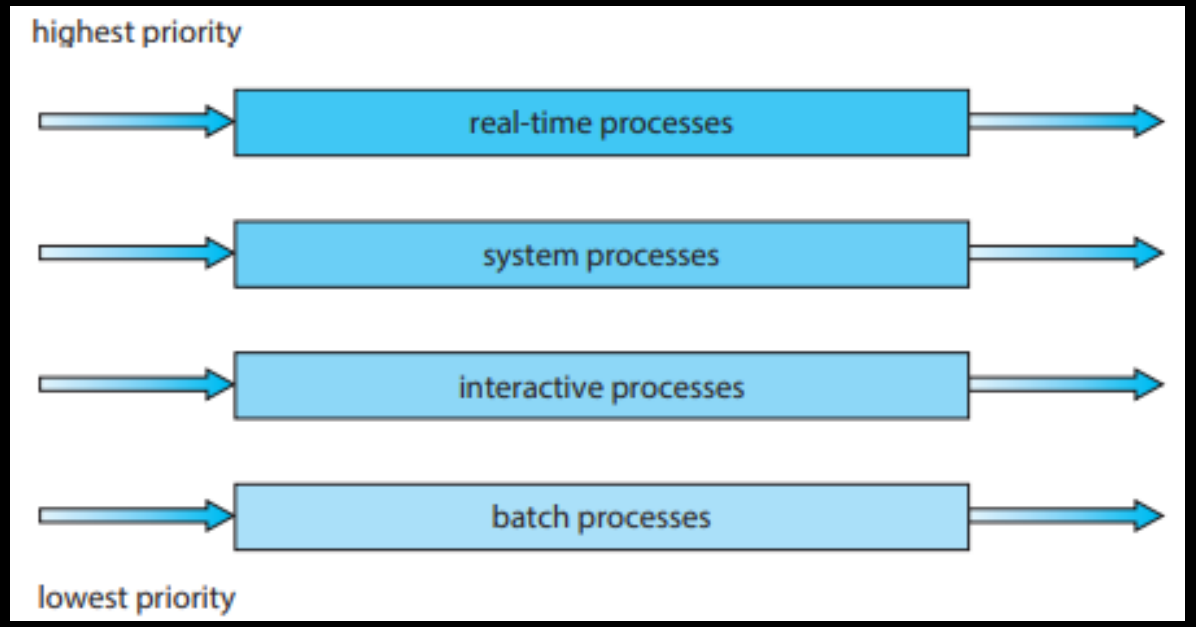
$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \cdots + (1 - \alpha)^j \alpha t_{n-j} + \cdots + (1 - \alpha)^{n+1} \tau_0.$$

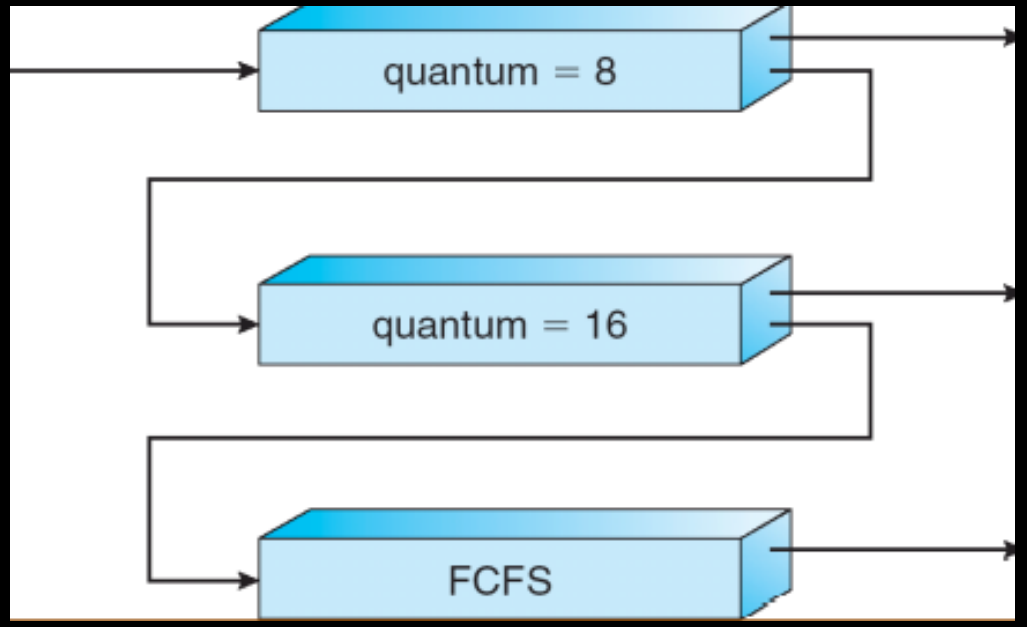
Dan karena pada umumnya $\alpha < 1$, maka $1 - \alpha < 1$ (atau $0 < \alpha$), sehingga setiap successive termnya punya weight yang lebih kecil daripada pendahulunya.

Editor note: biasanya sih, yang dikasih di soal asumsinya udah next burst time yang gak perlu dihitung pake rumus ini. Paling formula ini dipake kalo dibilang secara eksplisit harus diprediksi (dikasitau τ_n dan α).

Multilevel Queue & Multilevel Feedback Queue

- Multilevel Queue: ready queue dibagi menjadi beberapa queue, dimana tiap queue bisa memiliki algoritma yang berbeda. Contohnya ready queue menjadi dua queue yg terpisah dengan algoritma yang berbeda, foreground (RR) dan background (FCFS). Terdapat scheduling diantara masing-masing queue, biasanya dalam bentuk fixed priority scheduling (prioritas absolut) yang memungkinkan starvation, atau time slice, misalnya 80% CPU time untuk foreground dan 20% untuk background.
- Multilevel Feedback Queue: proses dapat dipindahkan diantara beberapa queue, dimana aging dapat diimplementasikan disini. Scheduler ini didefinisikan dengan beberapa parameter: banyaknya queue, algoritma yang dipakai untuk setiap queue, metode yang dipakai untuk upgrade/demote sebuah proses, dan metode yang dipakai untuk menentukan queue yang dipakai oleh proses baru.



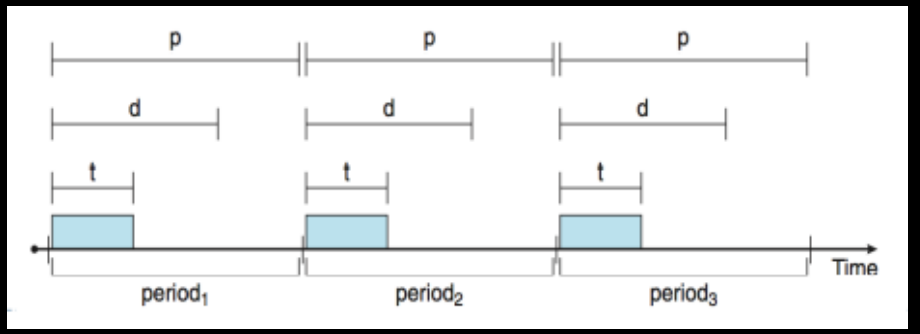


Thread Scheduling

- Ketika thread disupport, maka yang akan dijadwalkan adalah kernel-level threadnya, bukan proses.
- Untuk model many-to-one dan many-to-many, thread library menjadwalkan user level pada LWP* (Lightweight Process) yang tersedia. Skema ini dikenal sebagai Process Contention Scope (PCS), karena thread-thread pada proses yang sama berkompetisi untuk mendapatkan CPU.
- Untuk menentukan kernel-thread mana yang harus dijadwalkan ke CPU, kernel menggunakan system-contention scope (SCS). Disini, semua thread dalam system harus berkompetisi untuk mendapatkan jatah CPU. Model one-to-one hanya memakai SCS saja.

Editor's note: *LWP = struktur data mapping yang menghubungkan antara user dan kernel thread.

Real-Time CPU Scheduling

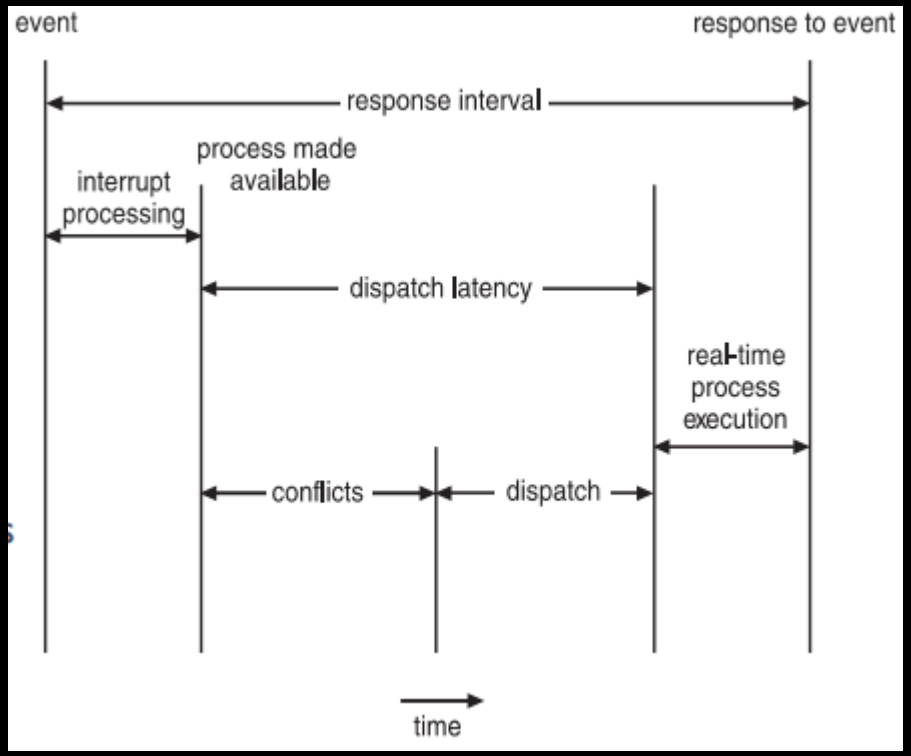


- Dalam kasus real-time, ada dua jenis system:

1. Soft real-time system: Tidak peduli kapan proses akan dijadwalkan.
2. Hard real-time system: Proses harus dijadwalkan sebelum deadline.

- Terdapat dua tipe latensi yang berdampak pada performa:

1. Interrupt latency: waktu sejak kedatangan interrupt hingga dimulainya interrupt service routine tersebut.
2. Dispatch latency



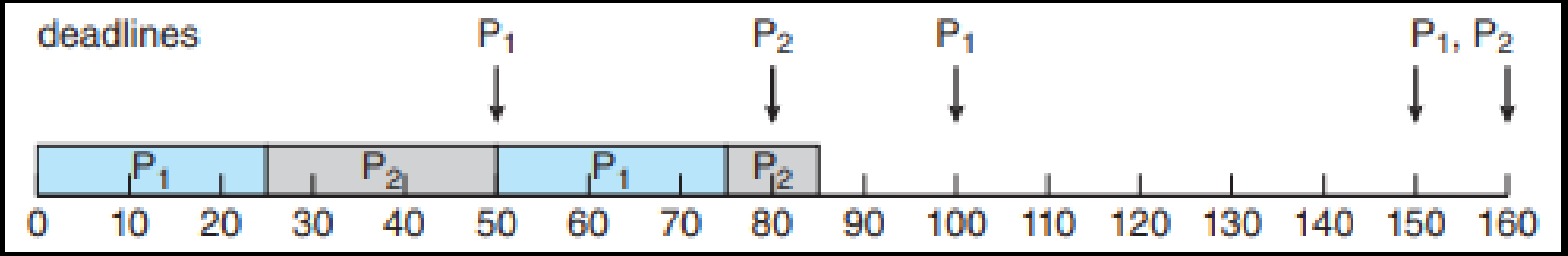
- Fase konflik pada dispatch latency disebabkan karena:

1. Preemption proses apapun yang sedang berjalan di kernel mode.
2. Melepaskan resource dari proses low-priority yang dibutuhkan proses high-priority.

- Untuk soft real-time, scheduler harus support preemptive, priority-based scheduling.
- Khusus untuk hard real-time, scheduler juga harus bisa menepati deadline.
- Proses juga mempunyai karakteristik baru: periodic p (membutuhkan CPU dalam interval konstan, tiap periode disebut periodic process), processing time t, dan deadline d, dimana $0 \leq t \leq d \leq p$, dengan rate periodiknya $1/p$.

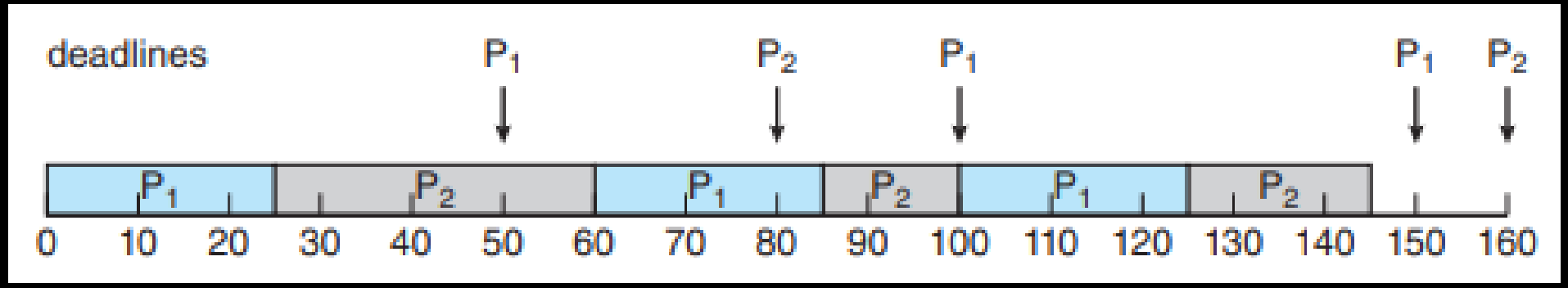
Rate Monotonic Scheduling

- Preempted, priority-based scheduling dimana prioritas didasarkan oleh inverse dari periodnya. Dengan kata lain, periode singkat = prioritas tinggi, periode lama = prioritas rendah.
- Asumsinya, processing time dari proses periodik sama untuk setiap CPU burst (setiap kali proses mendapatkan jatah CPU, durasi CPU burstnya sama).
- Kelemahan dari rate monotonic scheduling adalah tidak menjamin bahwa proses dapat dijadwalkan tepat deadline dan maksimum CPU utilization untuk N proses adalah $N(2^{1/N} - 1)$.
- Misalnya proses P_1 punya $p_1 = 50$ dan $t_1 = 25$. Sedangkan proses P_2, $p_2 = 80$ dan $t_2 = 35$. Pada periode pertama, P1 akan dijadwalkan hingga t = 25, dilanjutkan dengan P2 hingga t = 50, menyisakan waktu 10. P1 kemudian menyerobot P2 pada t = 50 karena sudah periode kedua hingga t = 50 + 25 = 75, kemudian dilanjutkan dengan sisa P2 hingga t = 85. Tetapi, deadline P2 adalah t = 80, sehingga P2 tak menepati deadline.



Earliest-Deadline-First (EDF) Scheduling

- Preempted, priority-based scheduling didasarkan oleh deadline (lebih cepat deadlinenya, lebih tinggi prioritasnya, dan sebaliknya). Prioritas ini dinamis seiring bertambahnya proses.
- Tak seperti rate-monotonic, EDF tidak mengharuskan prosesnya dalam bentuk periodic, maupun mengharuskan CPU time konstan per burst. Proses hanya diwajibkan untuk mengumumkan deadlinenya ke scheduler saat akan dijalankan.
- Secara teoritis, proses dapat dijadwalkan tepat waktu dengan CPU utilization 100%, tetapi dalam praktiknya, context switching antar proses dan interrupt handling membutuhkan waktu sehingga tak mungkin untuk mendapatkan utilisasi CPU sebesar 100%.
- Misalnya pada kasus sebelumnya, P1 memiliki deadline yang lebih cepat, sehingga akan dijadwalkan terlebih dahulu hingga t=25. Kemudian, P2 akan dijadwalkan hingga t=60. P1 tidak akan menyerobot P2 karena deadline P2 yang lebih dahulu dibandingkan deadline P1 selanjutnya.



Synchronization

Race condition, critical section, Peterson’s algo, hardware support, mutex lock, semaphore, classic problem, monitors

Race Condition

```
// Producer
while (true) {
        /* produce an item in next produced */
        while (counter == BUFFER_SIZE) ;
        /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++; // r=counter;r=r+1;counter=r
}
```

```
// Consumer
while (true) {
        while (counter == 0) ;
        /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--; // r=counter;r=r-1;counter=r
        /* consume the item in next consumed */
}
```

- Race condition = ketika beberapa proses mengakses dan memanipulasi data yang sama secara konkuren dan outcome yang diharapkan tergantung oleh urutan aksesnya.
- Misalnya pada contoh producer-consumer diatas, saat producer mengeksekusi r = counter; dan r = r + 1;, CPU dapat context switch tiba-tiba ke consumer, mengeksekusi r = counter dengan counter yang belum diubah dari producer.
- Dibutuhkan suatu sinkronisasi antar proses yang akan dibahas disini.

Critical Section Problem

- Pada system dengan n proses (p0, p1, …, pn-1), setiap proses punya critical section pada segmen kodenya jika proses mengubah variable yang sama, memperbarui table, menulis file, dst.
- Ketika sebuah proses berada pada critical section, tidak boleh ada proses lain yang juga berada pada critical section tersebut.
- Setiap proses harus meminta izin untuk memasuki critical section pada entry section, kemudian setelah critical section diakhiri dengan exit section, dan terakhir remainder section
- Permasalahan ini dapat diselesaikan jika 3 syarat ini seluruhnya memenuhi:

1. Mutual Exclusion (Mutex): Jika proses P_i sedang mengeksekusi critical section, tidak boleh ada proses lain yang juga mengeksekusi hal yang sama (setia/ga boleh selingkuh).
2. Progress: Jika tidak ada yang sedang mengeksekusi critical section dan ada beberapa proses yang ingin memasuki critical section, salah satu proses tersebut dipilih dan harus memasuki critical section tanpa jeda waktu yang tak terbatas.
3. Bounded waiting: Terdapat batasan maksimal untuk dapat mengakses critical section pada setiap proses, sehingga setiap proses pasti bisa mengeksekusi critical sectionnya (kebagian jatah sama rata).

Peterson's Algorithm

- Menyelesaikan permasalahan critical section untuk dua proses.
- Asumsi: instruksi load dan store bersifat atomic (tak bisa diinterrupt).
- Dua proses tsb share dua variable:
- int turn; // mengindikasikan giliran siapa yang bisa masuk ke critical section
- boolean flag[2]; // mengindikasikan proses siap untuk memasuki critical section

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);
          critical section
    flag[i] = false;
          remainder section
} while (true);
```

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;
   /* critical section */
   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = false;
   else
      waiting[j] = false;
   /* remainder section */
} while (true);
```

```
int compare_and_swap(int *value, int expected,
  int new_value) {
   int temp = *value;
   if (*value == expected)
      *value = new_value;
   return temp;
}
```

```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;

   return rv;
}
```

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */

      /* critical section */

   lock = false;

      /* remainder section */
} while (true);
```

```
do {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */
      /* critical section */
   lock = 0;
      /* remainder section */
} while (true);
```

Hardware Support

- Banyak system yang menyediakan hardware support untuk critical section.
- Solusinya terinspirasi dari locking: memproteksi critical section dengan lock (acquire lock, critical section, release lock, remainder section)
- Dalam uniprocessor (prosesor dengan 1 core), interrupt bisa dimatikan, jadi kode yang berjalan dapat dieksekusi tanpa pre-emption. Hal ini tidak efisien untuk system multiprosesor, dan OS yang pakai approach ini ngga scalable.
- Mesin modern sekarang mendukung instruksi hardware special yang bersifat atomik. Bentuk abstraksi instruksinya bisa berupa test_and_set() dan compare_and_swap().
- Dengan test_and_set atau compare_and_swap, kita dapat memperoleh mutex dengan deklarasi variable lock = false.
- Tetapi kedua instruksi tersebut tidak memenuhi kriteria bounded-waiting karena proses yang sama dapat memperoleh lock berulang kali sehingga terjadi starvation bagi proses lain. Hal ini dapat dicegah dengan melakukan pengecekan sirkuler dengan boolean waiting[i] yang mengidentifikasikan proses i ingin mengakses critical section.

Mutex Locks

- Mutex lock memberikan solusi high-level yang lebih simple untuk menyelesaikan permasalahan critical section.
- Dalam mutex lock, sebuah proses harus mendapatkan lock dengan acquire() sebelum memasuki critical section. Setelah selesai mengeksekusi critical section, proses harus melepaskan lock dengan release().
- Terdapat variable available yang mengindikasikan apakah lock tersedia apa tidak. Pemanggilan acquire dan release juga harus bersifat atomik, dibantu dengan instruksi hardware atomic.
- Permasalahan terbesar dalam mutex lock adalah terjadinya busy waiting, dimana proses yang ingin memasuki critical section ketika lock tidak tersedia harus looping/"spins" dalam acquire(). Kejadian ini juga bisa disebut sebagai spinlock. [busy waiting juga ada pada instruksi hardware atomik test_and_set atau compare_and_swap]

```
acquire() {
    while (!available)
        ; /* busy wait */
    available = false;;
}
release() {
    available = true;
}
```

```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (true);
```

Semaphore

- Semaphore S merupakan variable integer yang hanya bisa diakses melalui dua operasi atomik standar: wait() dan signal().
- Terdapat dua jenis semaphore: counting semaphore yang tidak dibatasi rangenya, dan binary semaphore yang hanya bisa bernilai 0 atau 1 (thus, sama dengan mutex lock)
- Semaphore bisa diimplementasikan tanpa busy waiting, dengan cara meletakkan proses ke waiting queue (block atau sleep) dan mengembalikan proses dari waiting queue ke ready queue (wakeup).
- Penggunaan semaphore dapat menyebabkan deadlock, starvation, dan priority inversion

```
wait (S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
signal (S) {
    S++;
}
```

```
Semaphore synch(0)
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

```
typedef struct{
    int value;
    struct process *list;
} semaphore;
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}
signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

Classical Problems of Synchronization

- Bounded Buffer problem: n buffer, tiap buffer cuma bisa menyimpan 1 item. Ada 3 semaphore: mutex (value 1), full (value 0), empty (value n)
- Readers-Writers problem: data set dishare ke beberapa proses konkuren, reader cuma bisa baca, tapi writer bisa baca maupun tulis. Permasalahannya adalah mengizinkan beberapa reader untuk membaca dalam waktu yang sama, tetapi hanya satu writer yang bisa akses shared data dalam waktu yang sama. Share datanya berupa data setnya, integer read_count (inisialisasi 0), semaphore rw_mutex (untuk lock writer, value 1), semaphore mutex (untuk manipulasi read_count di reader, inisialisasi 1).
- Dining philosopher: Terdapat N philosopher dengan N chopstick dan bowl of rice (dataset). Tiap philosopher harus mengambil dua chopstick yang bersebelahan untuk makan, kemudian mengembalikan kedua chopstick untuk dipakai philosopher lain. Permasalahan yang terjadi adalah penyalahgunaan semaphore sehingga menyebabkan deadlock hingga starvation.

```
do {
    ...
   /* produce an item in next_produced */
    ...
   wait(empty);
   wait(mutex);
    ...
   /* add next produced to the buffer */
    ...
   signal(mutex);
   signal(full);
} while (true);
```

```
do {
   wait(full);
   wait(mutex);
    ...
   /* remove an item from buffer to next_consumed */
    ...
   signal(mutex);
   signal(empty);
    ...
   /* consume the item in next consumed */
    ...
} while (true);
```

```
do {
   wait(rw_mutex);
    ...
   /* writing is performed */
    ...
   signal(rw_mutex);
} while (true);
```

```
do {
   wait(mutex);
   read count++;
   if (read_count == 1)
      wait(rw_mutex);
   signal(mutex);
   ...
   /* reading is performed */
   ...
   wait(mutex);
   read count--;
   if (read_count == 0)
      signal(rw_mutex);
   signal(mutex);
} while (true);
```

```
do  {
       wait (chopstick[i] );
       wait (chopStick[ (i + 1) % 5] );

              //  eat

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

              //  think

} while (TRUE);
```

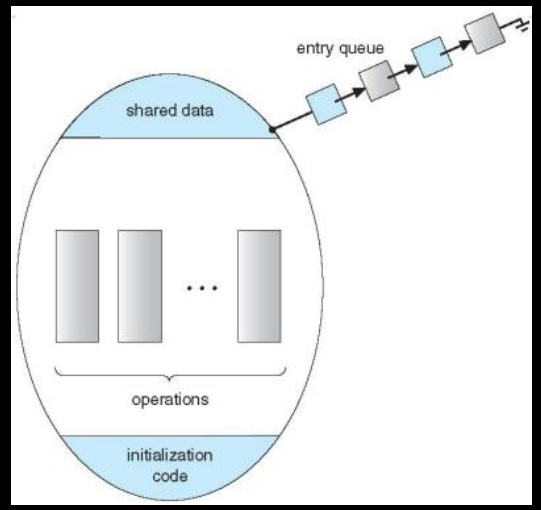
Monitors

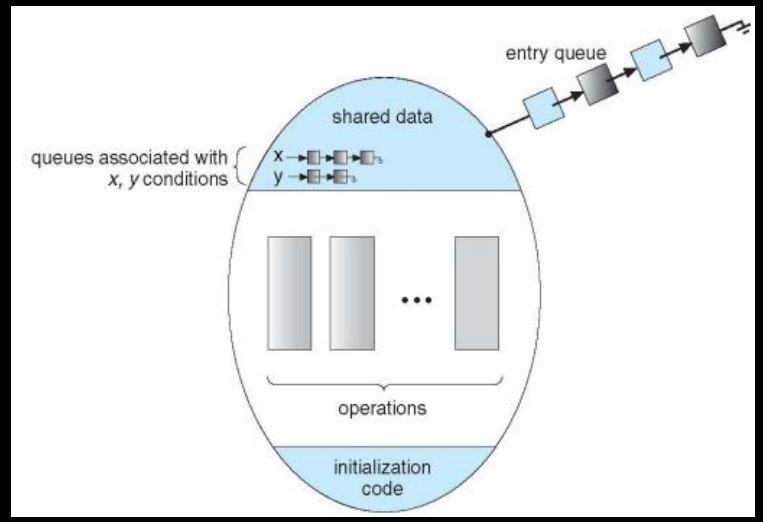
- Abstraksi high-level untuk mekanisme sinkronisasi yang nyaman dan efektif, tetapi tidak cukup powerful dalam memodelkan beberapa skema sinkronisasi.
- Merupakan bentuk ADT, dengan variable internal hanya bisa diakses dalam prosedur di dalam ADT.
- Hanya boleh satu proses yang aktif dalam satu monitor pada suatu waktu.
- Monitor dapat menggunakan condition variables, condition x, y; dengan tiap kondisi memiliki operasi wait() yang memberhentikan operasi, dan signal() yang melanjutkan operasi yang memanggil wait() jika ada.
- Jika proses P memanggil x.signal() dengan Q pada state x.wait(), dapat terjadi antara: signal and wait (P wait hingga Q keluar dari monitor atau menunggu kondisi lain) atau signal and continue (Q wait hingga P keluar dari monitor atau menunggu kondisi lain)
- Terdapat conditional-wait dimana x.wait(c) dengan c = priority number (c paling kecil akan dijadwalkan selanjutnya).

```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (…) { …. }

  procedure Pn (…) {……}

     Initialization code (…) { … }
  }
}
```






```
monitor ResourceAllocator
{
  boolean busy;
  condition x;
  void acquire(int time) {
            if (busy)
               x.wait(time);
            busy = TRUE;
  }
  void release() {
            busy = FALSE;
            x.signal();
  }
initialization code() {
   busy = FALSE;
  }
}
```

Deadlock

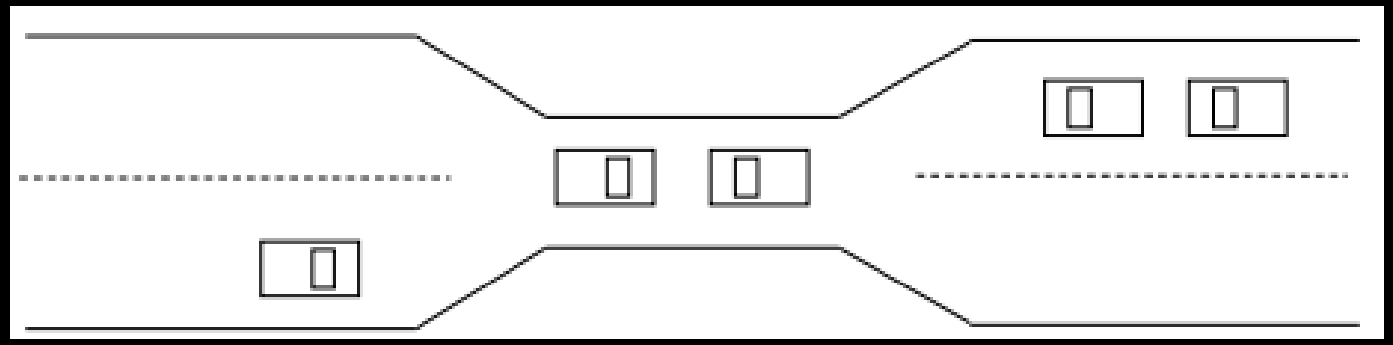
Definition & system model, characteristics, RAG, handling: [prevention, avoidance: safety/bankers algo/safe state, detection: algo/recovery.

Definisi & Model

- Sebuah set berupa blocked processes: tiap proses memegang resource dan saling menunggu untuk mendapatkan resource yang dipegang oleh proses dalam set.
- Contoh:

1. Dalam system yang punya 2 tape drives, P1 dan P2 memegang 1 tape drive dan saling membutuhkan tape lainnya.
2. Semaphore A dan B yang awalnya diinisialisasi dgn 1, pada P0 terdapat wait(A); wait(B); dan pada P1 terdapat wait(B); wait(A); sehingga saling menunggu.

- Dalam pemodelan system, terdapat tipe resource $R_1, R_2, \ldots, R_m$ (CPU cycle, memory space, I/O devices) dan tiap resource R_i memiliki W_i instances. Tiap proses menggunakan resource dengan sequence request – use – release.

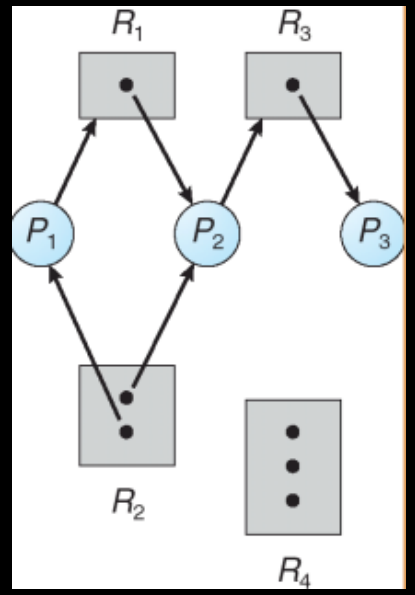


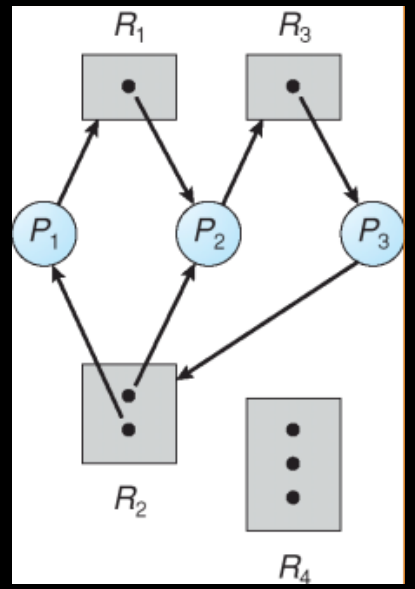
Karakteristik

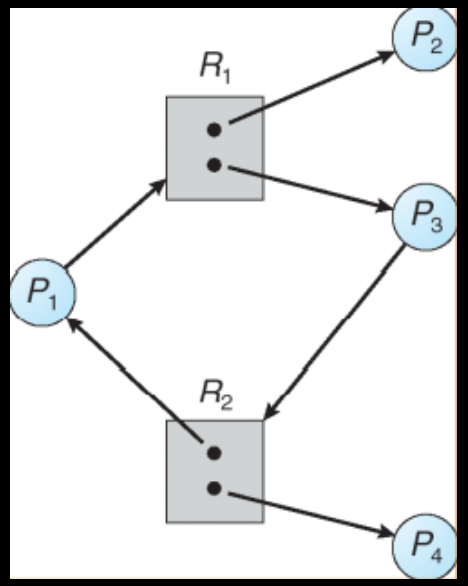
1. Mutual Exclusion (Mutex): Jika proses P_i sedang mengeksekusi critical section, tidak boleh ada proses lain yang juga mengeksekusi hal yang sama (setia/ga boleh selingkuh).
2. Hold and wait: Sebuah proses yang memegang paling sedikit 1 resource menunggu untuk mendapatkan resource lain yang sedang dipegang oleh porses lain.
3. No preemption: Resource hanya bisa dilepaskan oleh proses yang memegangnya secara voluntir setelah selesai menggunakan resource.
4. Circular wait: Himpunan waiting processes $\{P_0, P_1, \ldots, P_n, P_0\}$ sedemikian sehingga P_0 menunggu resource yang dipegang P_1, P_1 menunggu resource yang dipegang P_2, …, P_{n-1} menunggu resource yang dipegang P_n, dan P_0 menunggu resource yang dipegang P_0.

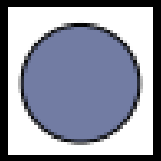
Resource-Allocation Graph (RAG)

- Himpunan simpul V dan sisi E, dengan V dibagi menjadi dua tipe,
  - $P = \{P_1, P_2, \dots, P_n\}$ yang merupakan himpunan semua proses dalam system.
  - $R = \{R_1, R_2, \dots, R_n\}$ yang merupakan himpunan semua tipe resource di system.
- Sisi request merupakan sisi berarah $P_1 \rightarrow R_1$
- Sisi assignment merupakan sisi berarah $R_1 \rightarrow P_1$
- Beberapa symbol dalam RAG dapat dilihat disamping.
- Beikut beberapa contoh RAG (no deadlock, with deadlock, cycle with no deadlock)

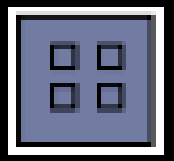




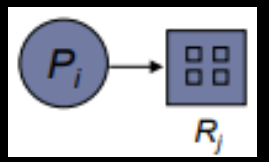




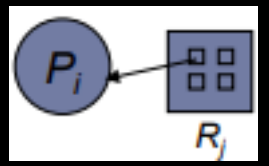
Proses



Resource Type dengan 4 Instance



P_i request instansi dari R_j (request edge)



P_i memegang instansi dari R_j (assignment edge)

- Jika graf tidak mempunyai cycle, maka tidak ada deadlock.
- Jika graf punya cycle:
  - Jika hanya ada satu instance per resource type, pasti terjadi deadlock
  - Jika ada lebih dari satu instance per resource type, mungkin terjadi deadlock

Handling

- Deadlock Prevention: Memastikan system tidak akan pernah memenuhi salah satu kriteria deadlock tanpa informasi tambahan untuk setiap proses tersebut. Biasanya hampir tidak mungkin untuk diimplementasikan.
- Deadlock Avoidance: Memastikan system tidak akan pernah memasuki unsafe state dimana deadlock dapat terjadi dengan informasi tambahan pada setiap proses.
- Deadlock Detection & Recovery: Memperbolehkan system memasuki state deadlock. Jika dideteksi terjadi deadlock, maka akan direcover oleh system.
- Dalam beberapa system operasi termasuk UNIX, permasalahan deadlock diabaikan dan berpura-pura kalau deadlock gak bakal pernah muncul di system.

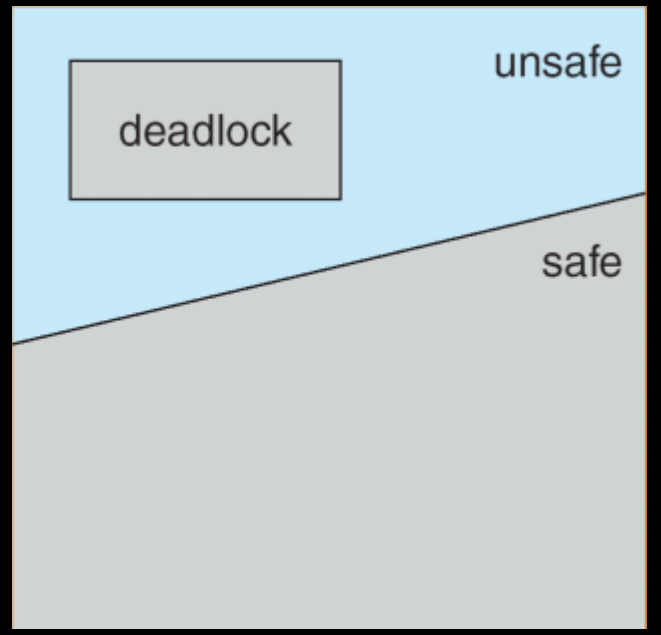
Deadlock Prevention

- Membatasi cara melakukan request:

1. Mutual Exclusion: tidak diharuskan untuk shareable resources, tetapi diharuskan untuk nonshareable resources.
2. Hold and wait: Harus menjamin bahwa kapanpun proses request sebuah resource, proses tersebut tidak memegang resource lain. Bisa dengan cara mengharuskan proses untuk request dan alokasi semua resourcenya sebelum dieksekusi (hanya sekali), atau proses hanya dibolehkan untuk request resource jika tidak memegang resource sama sekali. Hal ini bisa menyebabkan low resource utilization hingga starvation.
3. No pre-emption: Jika proses yang memegang resource apapun melakukan request resource lain yang tidak bisa langsung dialokasikan (terpaksa untuk wait), semua resource yang sedang dipegang akan dilepaskan (preempted), sehingga ketika resource yang direquest tersedia, proses harus mengambil kembali resource lama beserta yang direquest dalam satu request. Bisa juga proses tersebut mencari proses lain yang punya resource yang direquest dan juga wait untuk request resource baru, sehingga proses mengambil paksa resource dari process lain tersebut.
4. Circular wait: memberi nomor semua resource dan mengharuskan proses untuk merequest resource hanya dalam urutan menaik (atau menurun).

Deadlock Avoidance

- Mengharuskan proses memberi tahu maksimum resource yang dibutuhkan untuk setiap tipe resource.
- Algoritma deadlock avoidance secara dinamis menganalisis resource-allocation state untuk memastikan tidak ada kondisi circular-wait.
- Resource-allocation state didefinisikan sebagai banyaknya resource yang tersedia dan teralokasi, serta maximum demand dari proses.
- Safe state = Jika ada sebuah safe sequence dari semua proses pada system.
- Sequence $< P_1, P_2, \ldots, P_2 >$ adalah safe sequence jika untuk setiap P_i, resource masih bisa direquest P_i memenuhi resource yang tersedia + resource yang dipegang oleh semua P_j dengan $j < i$.
  - Jika kebutuhan resource P_j tidak tersedia secara langsung, maka P_i dapat menunggu hingga P_j selesai.
  - Ketika P_j selesai, P_i bisa mengambil resource yang dibutuhkan, mengeksekusi ditinya, mengembalikan resource yang dialokasi, dan terminate dirinya.
  - Ketika P_i terminate, P_{i+1} bisa mengambil resource yang dibutuhkan, dan seterusnya

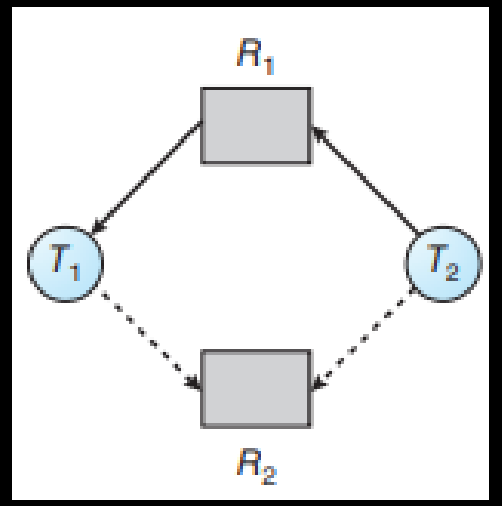


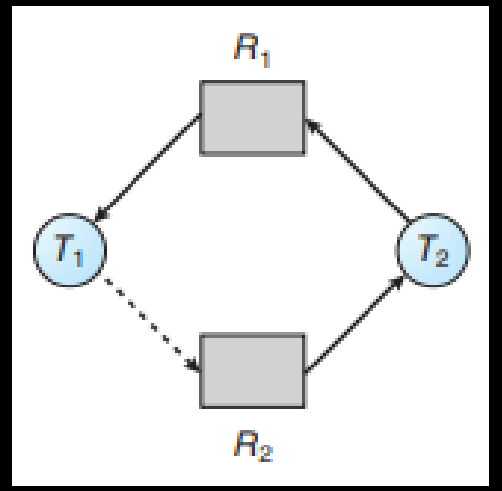
Jika system berada pada safe state, dijamin tak ada deadlock. Jika berada pada unsafe state, mungkin terjadi deadlock.

RAG Algorithm

Hanya bisa dipakai jika instance hanya 1 untuk setiap tipe resource

- Deadlock dapat dideteksi jika terdapat cycle dalam RAG.
- Claim edge $P_i \rightarrow R_j$ menandakan proses P_i yang mungkin request resource R_j, direpresentasikan dengan dashed line.
- Claim edge dikonversi menjadi request edge ketika proses request sebuah resource.
- Ketika sebuah resource dilepaskan oleh sebuah proses, assignment edge diubah menjadi claim edge
- Misalnya RAG disamping, jika T2 request R2, maka tidak akan diizinkan karena membentuk cycle pada RAG.





Banker's Algorithm

- Dapat dipakai untuk multiple instance pada tiap resource type.
- Terdiri dari dua algoritma: safety algorithm dan resource-request algorithm.
- Ketika proses request sebuah resource, dia mungkin harus menunggu.
- Ketika proses mendapatkan semua resourcenya, dia harus mengembalikannya dalam waktu yang terbatas.
- Misalnya n = banyaknya proses, m = banyaknya resource type:
  - Available: Array dengan panjang m. Available[i] = k berarti terdapat k instance di resource type R_i yang tersedia.
  - Max: n x m matrix. Max[i, j] = k berarti proses P_i mungkin meminta paling banyak k instance dari suatu resource type.
  - Allocation: n x m matrix. Allocation[i, j] = k berarti proses P_i mengalokasikan k instance dari R_j.
  - Need: n x m matrix. Need[i, j] = k berarti P_i mungkin membutuhkan k instance tambahan dari R_j untuk menyelesaikan tugasnya. $Need[i, j] = Max[i, j] - Allocation[i, j]$.

Safety Algorithm

1. $Work = Available$ (vector dgn panjang m)
   $Finish[i] = false$ (vector dgn panjang n utk setiap i = 0, 1, …, n-1)
2. Cari index i dimana
   $$\neg Finish[i] \wedge Need[i] \leq Work$$
3. Jika i ditemukan, lakukan:
   $$Work \mathrel{+}= Allocation$$
   $$Finish[i] = true$$
   Kemudian, kembali lagi ke step 2
4. Jika i tidak ditemukan, dan Finish[i] == true untuk semua i, maka system dalam keadaan safe state. Jika terdapat Finish[i] == false untuk salah satu i, maka system dalam keadaan unsafe state.

Algoritma ini memiliki kompleksitas sebesar $O(m \times n^2)$.

Resource-Request Algorithm

Diketahui sebuah request vector untuk proses P_i sebagai $Request[i]$

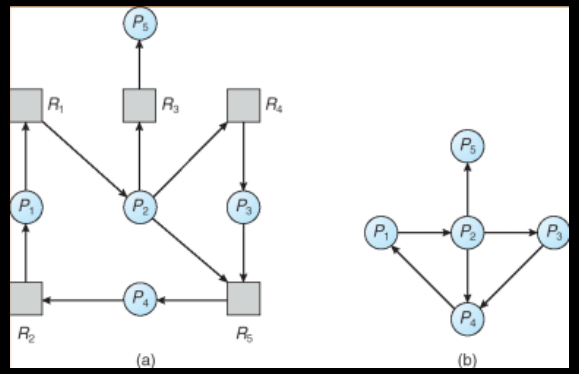
1. Jika $Request[i] \leq Need$, lanjut ke step 2. Jika tidak, throw error karena thread sudah memenuhi maksimum claim.
2. Jika $Request[i] \leq Available$, lanjut ke step 3. Jika tidak, P_i harus menunggu, karena resource belum tersedia.
3. Sistem memperbarui statenya sementara dengan pura-pura mengalokasikan resource yang direquest ke P_i:

$$Available -= Request[i]$$
$$Allocation[i] += Request[i]$$
$$Need[i] -= Request[i]$$

Kemudian, cek state yang telah diubah dengan safety algorithm. Jika statenya merupakan safe state, maka T_i diperbolehkan untuk mengalokasikan resourcenya. Jika statenya merupakan unsafe state, maka P_i harus menunggu untuk $Request[i]$ dan alokasi resource lamanya direstore.

Deadlock Detection

- Mengizinkan system untuk memasuki state deadlock, sembari menjalankan detection algorithm untuk mengetahui apakah terjadi deadlock apa tidak. Jika terjadi deadlock, maka akan dijalankan recovery scheme.
- Detection algorithm harus dipikirkan secara matang akan kapan dan seberapa sering untuk dijalankan. Bisa jadi terlalu banyak cycle pada RAG sehingga kita tidak tahu dari semua proses yang kena deadlock menjadi penyebab deadlock system.
- Jika semua resource type hanya memiliki 1 instance, detection algorithm dapat dijalankan dengan wait-for graph, dimana proses merupakan simpul dan terdapat sisi $P_i \rightarrow P_j$ jika P_i menunggu P_j. Akan dicek secara berkala apakah terdapat cycle (deadlock) pada wait-for graph tersebut.
- Gambar disamping merupakan (a) RAG dari model sebuah system dan (b) wait-for graph dari RAG di (a).
- Pada sebuah resource type dengan lebih dari 1 instance, diperlukan detection algorithm dengan struktur data berikut:
  - Available: Array dengan panjang m. Available[i] = k berarti terdapat k instance di resource type R_i yang tersedia.
  - Allocation: n x m matrix. Allocation[i, j] = k berarti proses P_i mengalokasikan k instance dari R_j.
  - Request: n x m matrix. Request[i, j] = k berarti proses P_i request k instance tambahan dari R_j.



Detection Algorithm

Jika terdapat lebih dari 1 instance untuk sebuah resource type

1. $Work = Available$ (vector dgn panjang m)
   $Finish[i] = false\ if\ Allocation[i] \neq 0\ else\ true$ (vector dgn panjang n utk setiap i = 0, 1, …, n-1)
2. Cari index i dimana
   $$\neg Finish[i] \wedge Request[i] \leq Work$$
3. Jika i ditemukan, lakukan:
   $$Work \mathrel{+}= Allocation$$
   $$Finish[i] = true$$
   Kemudian, kembali lagi ke step 2
4. Jika i tidak ditemukan, dan terdapat Finish[i] == false untuk salah satu i, maka system dalam keadaan deadlock. Lebih lengkapnya, jika $Finish[i] == false$, maka P_i dalam keadaan deadlock.

Algoritma ini memiliki kompleksitas sebesar $O(m \times n^2)$.

Recovery Scheme

- Process Termination: membatalkan semua proses yang deadlock atau membatalkan satu per satu semua proses yang deadlock hingga cycle pembentuk deadlock sudah hilang. Urutan pembatalan proses bisa dari prioritas dan berbagai factor lain: lamanya proses dan sisa waktu penyelesaian, resource yang telah digunakan proses, resource yang dibutuhkan proses, banyaknya proses yang harus dimatikan, dsb.
- Resource Preemption: Mengambil paksa resource (preemt) yang dibutuhkan proses yang deadlock dari proses lain hingga cycle pembentuk deadlock sudah hilang. Terdapat 3 issue yang perlu diperhatikan: pemilihan korban harus dengan meminimumkan cost, misalnya diperhitungkan dari banyaknya resource yang dipegang dan waktu yang sudah digunakan proses, kemudian rollback proses tersebut hingga mencapai safe state dan restart proses dari safe state susah untuk menentukan safe statenya dan membutuhkan banyak penyimpanan informasi untuk setiap proses yang berjalan. Selain itu, starvation, dimana resource yang di pre-empt selalu berasal dari proses yang sama, juga harus tidak boleh muncul,