

PERTEMUAN 3

PROSES DAN THREAD

Mata Kuliah	Sistem Operasi (SI-204) — 3 SKS (2 Teori, 1 Praktikum)
Pertemuan	3 dari 16
Sub-CPMK	Sub-CPMK-3: Mampu menjelaskan siklus hidup proses dan thread serta mengamati proses pada sistem operasi nyata (C2, P2).
CPMK Terkait	CPMK-1 dan CPMK-4
Indikator	• Menjelaskan state proses dan PCB • Menjelaskan context switch dan model thread • Mengelola proses dan service di Linux
Bentuk & Waktu	Problem-Based Learning mini (TM 2×50'); Praktikum (1×170'); Penugasan Terstruktur (2×60'); Belajar Mandiri (2×60')
Penilaian	Kuis dan Laporan Praktikum 3 (ketepatan konsep dan keterampilan perintah)
Referensi	[1] Bab 3–4; [2] Bab 2
Prasyarat	Pertemuan 1 (dual-mode, interupsi, timer) dan Pertemuan 2 (system call, boot, systemd sebagai PID 1); VM Ubuntu Server yang dapat diakses melalui SSH dari Praktikum 2

A. TUJUAN PEMBELAJARAN

Setelah mengikuti pertemuan ini, mahasiswa diharapkan mampu:

1. Membedakan program dan proses, serta menjelaskan isi ruang memori sebuah proses.
2. Menjelaskan lima state proses beserta peristiwa yang memindahkan proses dari satu state ke state lain.
3. Menjelaskan isi *Process Control Block* (PCB) dan fungsinya.
4. Menguraikan langkah *context switch* dan menjelaskan mengapa ia merupakan overhead.
5. Menjelaskan pembuatan dan pengakhiran proses (**fork**, **exec**, **wait**, **exit**) serta dua bentuk IPC dasar.
6. Membedakan proses dan thread, menyebutkan manfaat multithreading, dan membandingkan model many-to-one, one-to-one, dan many-to-many.
7. (Praktikum) Mengamati dan mengendalikan proses di Linux memakai **ps**, **top/htop**, **kill**, **nice**, dan mengelola service dengan **systemctl**.

Mengapa materi ini penting bagi mahasiswa SI?

Setiap aplikasi SI yang Anda bangun, misalnya web KRS, API pembayaran, atau basis data, pada akhirnya adalah **proses** di server. Ketika sistem lambat, tidak merespons, atau "menggantung", pertanyaan pertama seorang administrator adalah: *proses apa yang sedang berjalan, dalam state apa, dan siapa yang memakai CPU?* Materi ini memberi bahasa dan alat untuk menjawabnya.

B. KONSEP PROSES

B.1 Mengingat Kembali

Pada Pertemuan 1 kita belajar bahwa OS membagi CPU secara bergantian kepada banyak program menggunakan **interupsi timer**. Pada Pertemuan 2 kita melihat bahwa program meminta layanan kernel lewat **system call**. Sekarang kita menjawab pertanyaan yang lebih mendasar: *apa sebenarnya yang dibagi-bagi OS itu?* Jawabannya adalah **proses**.

B.2 Program vs Proses

Program adalah berkas di disk berisi instruksi, bersifat pasif, ibarat resep masakan di buku. **Proses** adalah program yang **sedang dieksekusi**, bersifat aktif, ibarat koki yang sedang memasak mengikuti resep itu, lengkap dengan bahan yang sedang diolah dan langkah resep yang sedang dikerjakan.

Satu program dapat menjadi banyak proses. Jika tiga mahasiswa membuka terminal dan masing-masing menjalankan **python3**, ada satu program tetapi tiga proses dengan data, posisi eksekusi, dan memori yang terpisah.

Aspek	Program	Proses
Sifat	Pasif (berkas di disk)	Aktif (sedang berjalan di memori)
Keberadaan	Ada selama berkasnya ada	Ada sejak dibuat sampai selesai
Sumber daya	Hanya memakai ruang disk	Memakai CPU, memori, berkas terbuka, dan lain-lain
Jumlah	Satu berkas	Dapat banyak proses dari satu program
Identitas	Nama berkas	<i>Process ID</i> (PID) yang unik

B.3 Isi Ruang Memori Sebuah Proses

Ketika OS memuat program menjadi proses, ia menyiapkan ruang memori yang terdiri dari empat bagian utama (Gambar 3.1):



Gambar 3.1 Dari program menjadi proses dan tata letak memorinya

- **Text:** instruksi program. Bagian ini hanya-baca.

- **Data:** variabel global dan statis (termasuk yang belum diberi nilai awal, disebut *bss*).
- **Heap:** memori yang diminta program saat berjalan, misalnya lewat `malloc()` di C atau `new` di Java. Tumbuh ke arah alamat yang lebih tinggi.
- **Stack:** variabel lokal, parameter fungsi, dan alamat kembali tiap pemanggilan fungsi. Pada arsitektur x86 tumbuh ke arah alamat yang lebih rendah.

Selain memori, OS juga mencatat **posisi eksekusi** (nilai *program counter*), **isi register CPU**, dan **daftar berkas yang sedang dibuka** proses itu. Semua catatan ini disimpan dalam PCB (Bagian D).

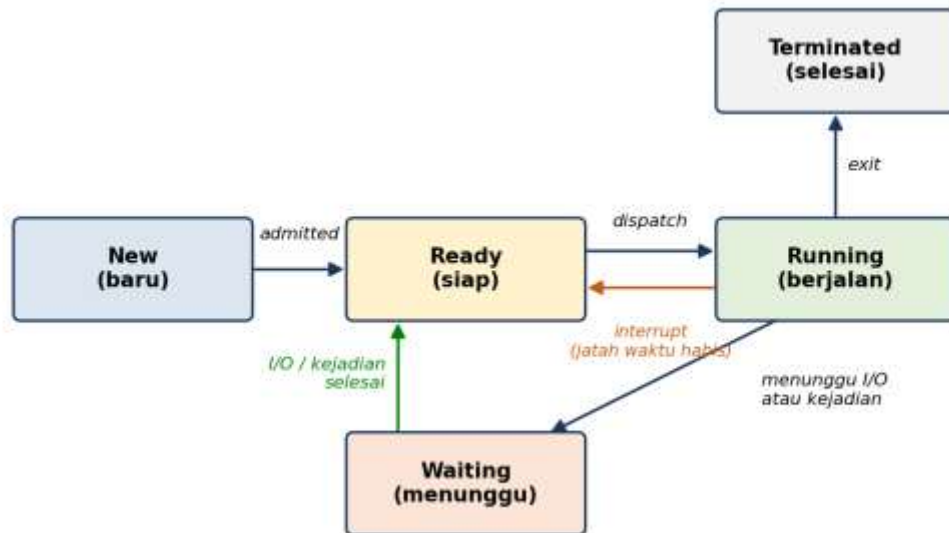
Kesalahpahaman umum

"Proses dan program sama saja." Keliru. Menutup jendela aplikasi mengakhiri **proses**, sedangkan **program** (berkasnya) tetap ada di disk. Menjalankan program dua kali menghasilkan dua proses yang terpisah dan tidak saling melihat memori satu sama lain.

C. STATE PROSES

Sebuah proses tidak selalu berjalan di CPU. Selama hidupnya, ia berpindah di antara beberapa **state** (keadaan). Model lima state yang umum dipakai adalah sebagai berikut.

State	Arti	Contoh
New	Proses sedang dibuat; sumber dayanya sedang disiapkan.	Pengguna baru menjalankan perintah
Ready	Siap berjalan dan menunggu giliran memakai CPU.	Antre karena CPU sedang dipakai proses lain
Running	Instruksinya sedang dieksekusi oleh CPU. Pada satu inti CPU hanya satu proses yang running pada satu waktu.	Menghitung rekap nilai
Waiting	Menunggu suatu kejadian, misalnya selesainya I/O atau sinyal, sehingga belum bisa memakai CPU.	Menunggu data dari disk atau jaringan
Terminated	Eksekusi selesai; OS membersihkan sumber dayanya.	Program keluar normal atau dihentikan



Gambar 3.2 Diagram state proses dan peristiwa perpindahannya

Perpindahan antarstate dipicu oleh peristiwa tertentu:

Perpindahan	Penyebab	Siapa yang bertindak
New → Ready	Proses selesai disiapkan (<i>admitted</i>)	OS
Ready → Running	Penjadwal memilih proses ini (<i>dispatch</i>)	Penjadwal CPU (dipelajari di Pertemuan 4)
Running → Ready	Jatah waktu habis (interupsi timer) atau ada proses lain yang lebih penting (<i>preemption</i>)	Kernel, dipicu timer (ingat Pertemuan 1)
Running → Waiting	Proses meminta I/O atau menunggu kejadian	Proses itu sendiri, lewat system call
Waiting → Ready	I/O atau kejadian yang ditunggu selesai (lewat interupsi)	Kernel
Running → Terminated	Proses memanggil <i>exit</i> atau dihentikan	Proses atau kernel

Perhatikan satu hal penting: proses yang selesai menunggu **tidak langsung berjalan**, melainkan kembali ke state *Ready* dan antri lagi. Itulah sebabnya aplikasi yang sudah menerima datanya masih bisa terasa lambat jika CPU sedang ramai.

C.1 State Proses pada Linux

Linux memakai nama state yang sedikit berbeda. Kolom **STAT** pada perintah **ps** menampilkan huruf-huruf berikut, yang akan Anda temui langsung di praktikum.

Kode	Nama	Arti praktis
R	Running / runnable	Sedang berjalan atau siap berjalan (gabungan <i>Running</i> dan <i>Ready</i>)
S	Interruptible sleep	Menunggu kejadian; dapat dibangunkan oleh sinyal. Mayoritas proses ada di state ini
D	Uninterruptible sleep	Menunggu I/O (umumnya disk atau jaringan) dan tidak dapat diinterupsi , bahkan oleh kill -9 , sampai I/O selesai

Kode	Nama	Arti praktis
T	Stopped	Dihentikan sementara, misalnya dengan Ctrl+Z atau sinyal SIGSTOP
Z	Zombie	Sudah selesai, tetapi catatannya masih tertinggal karena induknya belum membaca status keluarnya

Beberapa huruf tambahan dapat menyertai, misalnya **s** (pemimpin sesi), **l** (memiliki banyak thread), **+** (berada di latar depan terminal), dan **<** atau **N** (prioritas tinggi atau rendah). Sehingga **Ssl** dibaca: sleeping, pemimpin sesi, multithread.

C.2 Zombie dan Orphan

Proses zombie adalah proses yang sudah selesai tetapi masih tercatat di tabel proses karena induknya belum memanggil `wait()` untuk mengambil status keluarnya. Zombie **tidak memakai CPU dan hampir tidak memakai memori**, tetapi menghabiskan satu entri PID. Menjalankan `kill` pada zombie tidak berpengaruh karena ia memang sudah mati. Cara membereskannya adalah membuat induknya memanggil `wait()` atau menghentikan induknya sehingga proses `init/systemd` mengambil alih dan membersihkannya.

Proses orphan adalah proses yang induknya selesai lebih dulu. Proses ini otomatis diadopsi oleh PID 1 (`systemd`) sehingga tidak dibiarkan tanpa induk.

Konteks SI

Zombie dalam jumlah kecil tidak berbahaya, tetapi ribuan zombie menandakan **bug pada program induk** (misalnya server yang membuat proses anak tanpa pernah memanggil `wait()`). Tanpa dibereskan, tabel PID dapat penuh dan server tidak bisa membuat proses baru, sehingga aplikasi terlihat "hidup" tetapi gagal melayani permintaan baru.

Kesalahpahaman umum

"Proses state D harus dimatikan dengan kill -9." Tidak bisa. State D berarti proses sedang menunggu I/O di dalam kernel. Biasanya masalahnya ada pada disk atau jaringan (misalnya NFS tidak merespons), bukan pada proses itu. Periksa perangkat penyimpanannya.

D. PROCESS CONTROL BLOCK (PCB)

Bagaimana OS bisa menghentikan proses di tengah jalan lalu melanjutkannya persis dari titik yang sama? Jawabannya: OS menyimpan semua informasi penting sebuah proses dalam struktur data di memori kernel bernama **Process Control Block (PCB)**. Setiap proses memiliki satu PCB, ibarat kartu catatan pasien di rumah sakit: siapa pasiennya, kondisinya, dan tindakan terakhir yang dilakukan.

Isi PCB	Fungsi
Identitas proses (PID, PID induk, pemilik)	Membedakan proses dan menentukan hak aksesnya
State proses	New, ready, running, waiting, atau terminated
Program counter	Alamat instruksi berikutnya yang akan dieksekusi
Register CPU	Menyimpan nilai register agar dapat dipulihkan saat proses lanjut
Informasi penjadwalan	Prioritas, nilai nice, riwayat pemakaian CPU

Isi PCB	Fungsi
Informasi manajemen memori	Tabel halaman atau batas ruang alamat proses
Informasi I/O dan berkas	Daftar berkas yang terbuka (<i>file descriptor</i>) dan perangkat yang dipakai
Informasi akuntansi	Total waktu CPU, batas sumber daya, waktu mulai

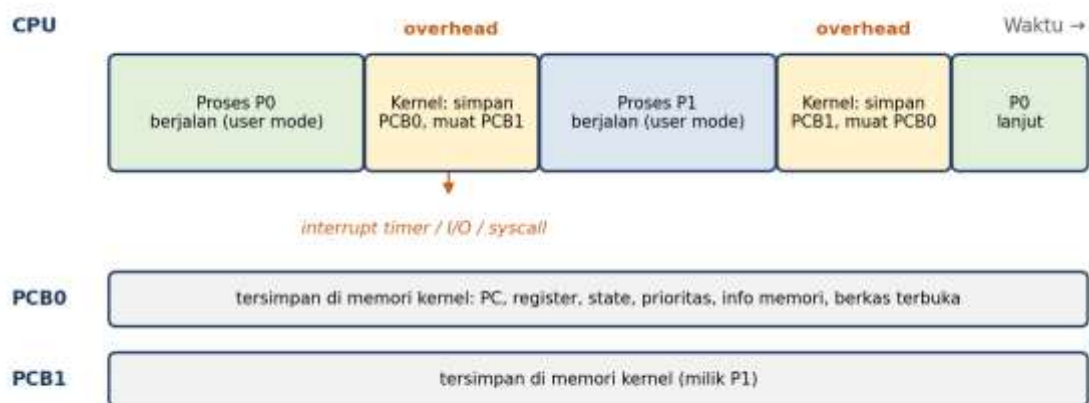
Di Linux, PCB diwujudkan sebagai struktur `task_struct` di dalam kernel. Sebagian isinya dapat Anda intip lewat berkas virtual `/proc/<PID>/` (misalnya `/proc/<PID>/status`), yang akan kita pakai saat praktikum.

Setiap proses juga memiliki **PID** unik dan **PPID** (PID induknya). Proses pertama setelah boot adalah PID 1 (`systemd`), dan semua proses lain adalah keturunannya sehingga membentuk **pohon proses**.

E. CONTEXT SWITCH

E.1 Apa yang Terjadi Saat CPU Berpindah Proses

Agar banyak proses tampak berjalan bersamaan, CPU berpindah dari satu proses ke proses lain dengan sangat cepat. Perpindahan ini disebut **context switch**. *Context* adalah "keadaan" proses: nilai register, program counter, dan informasi memori. Langkah-langkahnya (Gambar 3.3):



Gambar 3.3 Context switch antara proses P0 dan P1

1. Sebuah peristiwa memicu kernel: **interupsi timer** (jatah waktu habis), proses meminta I/O lewat **system call** dan harus menunggu, atau interupsi dari perangkat yang membangunkan proses yang lebih penting.
2. Kernel **menyimpan** konteks proses lama (P0) ke PCB-nya, termasuk register dan program counter.
3. Penjadwal memilih proses berikutnya (P1).
4. Kernel **memuat** konteks P1 dari PCB-nya, termasuk mengganti ruang alamat memori yang dipakai CPU.
5. CPU melanjutkan P1 persis dari titik terakhir ia berhenti. P1 tidak "sadar" pernah dihentikan.

E.2 Context Switch Adalah Overhead

Selama proses penyimpanan dan pemuatan, CPU **tidak mengerjakan pekerjaan berguna bagi pengguna mana pun**. Itulah sebabnya context switch disebut *overhead*. Ada juga biaya tidak langsung: setelah berpindah, isi *cache* CPU lebih banyak berisi data proses lama sehingga proses

baru awalnya lebih sering "meleaset" membaca cache. Lama satu context switch bergantung pada perangkat keras dan OS, umumnya berorde mikrodetik.

Akibatnya ada *trade-off*: jika jatah waktu terlalu pendek, sistem tampak sangat responsif tetapi banyak waktu terbuang untuk berpindah. Jika terlalu panjang, overhead kecil tetapi respons terasa lambat. Penjadwal CPU (Pertemuan 4) menyeimbangkan hal ini.

Linux mencatat jumlah context switch tiap proses. Pada `/proc/<PID>/status` ada dua baris: `voluntary_ctxt_switches` (proses melepas CPU sendiri karena menunggu, misalnya I/O) dan `nonvoluntary_ctxt_switches` (proses dipaksa melepas CPU karena jatah waktunya habis). Proses yang banyak menunggu didominasi *voluntary*, sedangkan proses yang menghitung terus-menerus didominasi *nonvoluntary*. Anda akan membuktikannya di praktikum.

F. OPERASI PADA PROSES

F.1 Membuat Proses: `fork`, `exec`, `wait`

Pada UNIX/Linux, proses baru dibuat lewat system call `fork()`. Proses induk **menggandakan** dirinya menjadi proses anak. Setelah `fork()` ada dua proses yang hampir identik, dan keduanya melanjutkan dari titik yang sama. Perbedaannya ada pada **nilai kembalian** `fork()`:

- pada **proses anak**, `fork()` mengembalikan `0`;
- pada **proses induk**, `fork()` mengembalikan PID anak;
- jika gagal (misalnya tabel proses penuh), mengembalikan `-1`.

Anak biasanya kemudian memanggil `exec()` untuk **mengganti isi memorinya** dengan program lain, sedangkan induk memanggil `wait()` untuk menunggu anak selesai. Untuk efisiensi, Linux memakai teknik *copy-on-write*: memori induk tidak langsung disalin seluruhnya, melainkan dibagi dahulu dan baru disalin ketika salah satu pihak mengubahnya.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void) {
    pid_t pid = fork();
    if (pid == 0) {
        execlp("ls", "ls", "-l", NULL);
        _exit(1);
    } else if (pid > 0) {
        wait(NULL);
        printf("Anak selesai\n");
    }
    return 0;
}
```

Iniilah yang dilakukan **shell** setiap kali Anda mengetik perintah: shell `fork()` dirinya, anak menjalankan `exec()` untuk program yang diminta, dan shell `wait()` sampai selesai. Jika perintah diakhiri `&`, shell tidak menunggu sehingga Anda langsung mendapat prompt kembali.

Pada Windows, `CreateProcess()` menggabungkan pembuatan proses dan pemuatan programnya sekaligus. Konsepnya sama (proses induk membuat proses anak), tetapi antarmukanya berbeda.

F.2 Mengakhiri Proses dan Sinyal

Proses berakhir secara normal dengan `exit()`, dengan mengirim kode status (0 berarti sukses) yang kemudian dibaca induknya lewat `wait()`. Proses juga dapat berakhir karena **sinyal**, yaitu notifikasi singkat yang dikirim kernel atau proses lain. Sinyal yang perlu Anda kenal:

Sinyal	No.	Arti dan penggunaan
SIGTERM	15	Permintaan agar proses berhenti dengan sopan. Proses dapat menangkapnya lalu merapikan pekerjaan (menutup berkas, menyimpan data). Ini sinyal bawaan perintah kill .
SIGKILL	9	Penghentian paksa oleh kernel. Tidak dapat ditangkap atau diabaikan , sehingga proses tidak sempat merapikan apa pun.
SIGINT	2	Interupsi dari keyboard (Ctrl+C).
SIGSTOP / SIGCONT	19 / 18	Menghentikan sementara dan melanjutkan proses. SIGSTOP tidak dapat ditangkap.
SIGHUP	1	Awalnya "sambungan terminal terputus". Banyak layanan memakainya sebagai perintah "baca ulang konfigurasi".

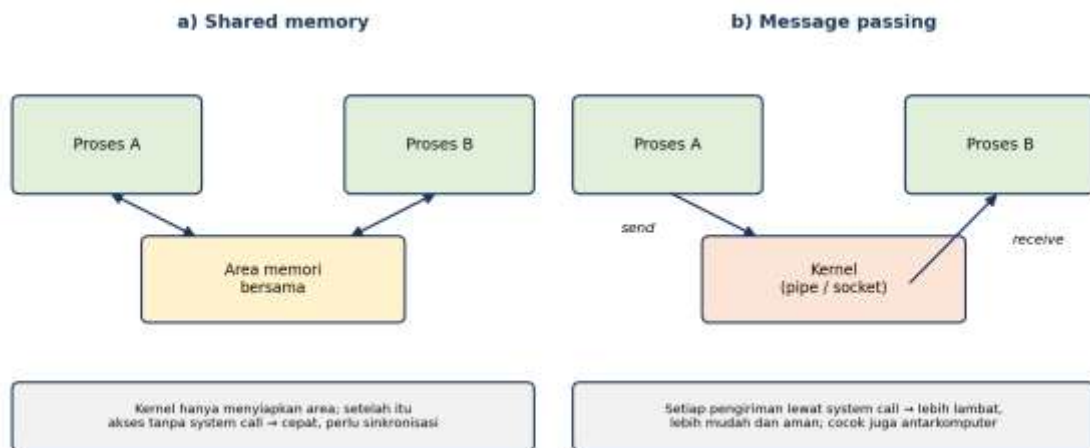
Praktik yang baik

Urutan yang aman ketika menghentikan proses: **SIGTERM** dulu, tunggu beberapa detik, baru **SIGKILL** jika tetap tidak berhenti. Memulai dengan **kill -9** berisiko merusak data, misalnya basis data yang sedang menulis atau berkas yang setengah tersimpan.

Nomor sinyal di atas berlaku untuk Linux pada arsitektur x86 dan ARM. Lihat daftar lengkapnya dengan **kill -l** atau **man 7 signal**.

G. KOMUNIKASI ANTARPROSES (IPC) DASAR

Proses dirancang **terisolasi**: tiap proses punya ruang memori sendiri sehingga satu proses tidak dapat membaca atau merusak memori proses lain. Isolasi ini bagus untuk keamanan dan keandalan, tetapi kadang proses perlu bekerja sama, misalnya server web meneruskan permintaan ke aplikasi, lalu aplikasi bertanya ke basis data. Untuk itu OS menyediakan **Inter-Process Communication (IPC)** dengan dua pendekatan dasar (Gambar 3.4).



Gambar 3.4 Dua pendekatan IPC: shared memory dan message passing

Aspek	Shared memory	Message passing
Cara kerja	Kernel menyiapkan area memori yang dipakai bersama; proses membaca dan menulis langsung	Proses saling mengirim dan menerima pesan lewat kernel
Kecepatan	Cepat setelah disiapkan (tanpa system call tiap akses)	Lebih lambat (tiap pengiriman lewat system call)

Aspek	Shared memory	Message passing
Kerumitan	Programmer harus mengatur sinkronisasi sendiri agar tidak terjadi tabrakan (Pertemuan 5)	Lebih sederhana dan aman; kernel yang mengatur antrean
Jangkauan	Hanya satu komputer	Satu komputer atau antarkomputer (socket)
Contoh di Linux	<code>shm_open()</code> , <code>mmap()</code>	<i>pipe</i> , <i>named pipe</i> (FIFO), <i>socket</i> , <i>message queue</i>

G.1 Contoh: Pipe pada Shell

Tanda `|` pada shell membuat **pipe**: keluaran satu proses menjadi masukan proses lain. Perintah berikut membuat **dua proses** yang dihubungkan satu pipe di dalam kernel.

```
ls /etc | wc -l
# proses 1 (ls) menulis daftar berkas ke pipe
# proses 2 (wc -l) membaca dari pipe dan menghitung barisnya
```

Socket adalah bentuk message passing yang juga bekerja antarkomputer. Inilah dasar komunikasi antara browser dan server web, atau antara aplikasi dan basis data. Detail tentang jaringan dipelajari di mata kuliah lain; di sini cukup dipahami bahwa socket adalah IPC.

Konteks SI

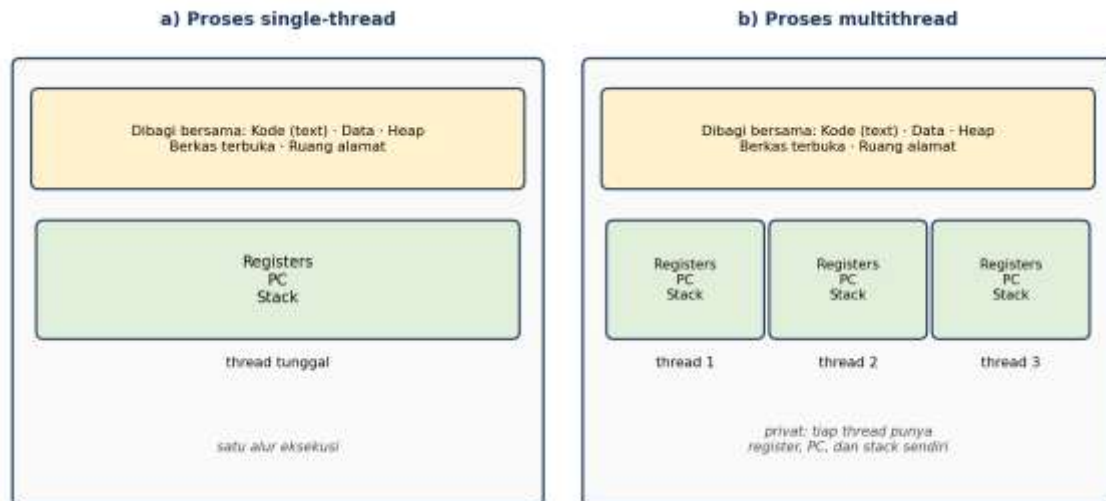
Arsitektur aplikasi web umumnya terdiri dari beberapa proses yang berbicara lewat IPC: Nginx (proses web server) meneruskan permintaan ke aplikasi (Gunicorn atau Node.js) melalui **socket**, dan aplikasi mengakses PostgreSQL juga lewat socket. Memahami IPC membantu Anda mendiagnosis "aplikasi hidup tetapi tidak dapat terhubung ke basis data".

H. THREAD

H.1 Mengapa Perlu Thread?

Bayangkan server aplikasi KRS yang membuat **satu proses baru** untuk setiap mahasiswa yang mengakses. Setiap proses butuh ruang memori sendiri dan pembuatannya relatif mahal. Selain itu, proses-proses itu tidak mudah berbagi data (misalnya cache nilai mata kuliah). Solusinya adalah **thread**.

Thread adalah satuan eksekusi di dalam proses. Satu proses dapat memiliki beberapa thread yang **berbagi kode, data, heap, dan berkas terbuka**, tetapi masing-masing memiliki **register, program counter, dan stack sendiri** (Gambar 3.5). Karena berbagi ruang memori, thread lebih ringan dibuat dan lebih mudah bertukar data dibanding proses terpisah.



Gambar 3.5 Proses single-thread dan multithread

Aspek	Proses	Thread
Ruang memori	Terpisah, terisolasi	Dibagi dengan thread lain dalam proses yang sama
Biaya pembuatan	Lebih besar	Lebih kecil
Biaya context switch	Lebih besar (ruang alamat ikut berganti)	Lebih kecil antarthread dalam satu proses
Komunikasi	Perlu IPC	Langsung lewat memori bersama (perlu sinkronisasi)
Dampak kegagalan	Crash satu proses umumnya tidak menjatuhkan proses lain	Crash satu thread dapat menjatuhkan seluruh proses

H.2 Manfaat Multithreading

- **Responsivitas:** aplikasi tetap tanggap karena thread lain dapat bekerja saat satu thread menunggu (misalnya UI tetap hidup saat berkas diunduh).
- **Berbagi sumber daya:** thread berbagi memori sehingga data tidak perlu disalin.
- **Ekonomis:** membuat dan berpindah antarthread lebih murah daripada antarproses.
- **Pemanfaatan multicore:** pada CPU multicore, beberapa thread benar-benar dapat berjalan **paralel** pada inti yang berbeda.

Konkurensi vs Paralelisme

Konkurensi: banyak tugas berkembang secara bergantian (bisa pada satu inti, dengan berpindah cepat). **Paralelisme:** banyak tugas benar-benar berjalan pada saat yang sama (butuh lebih dari satu inti). Sistem konkuren belum tentu paralel.

Peringatan

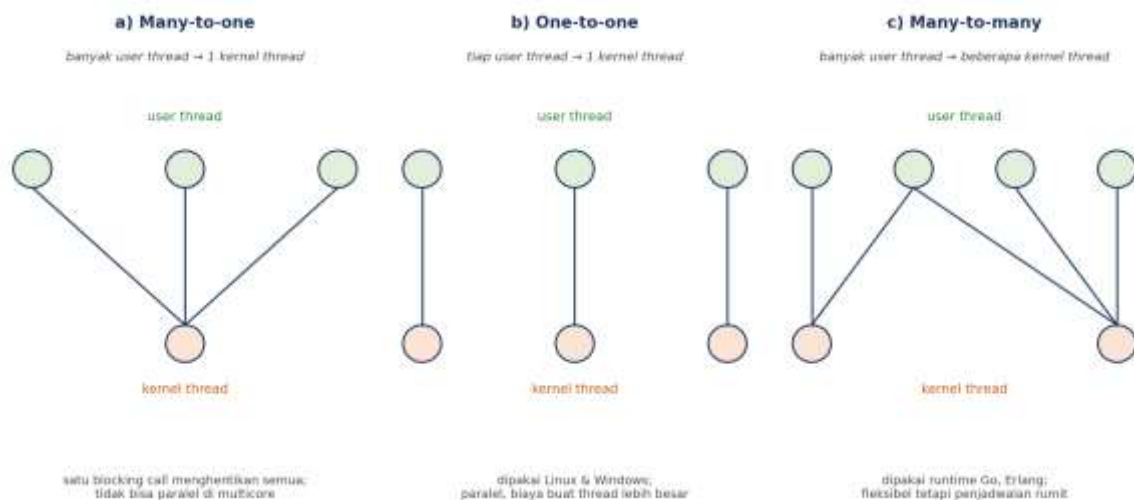
Karena thread berbagi memori, dua thread yang mengubah data yang sama tanpa koordinasi dapat menghasilkan hasil yang salah (*race condition*). Ini topik Pertemuan 5. Untuk sekarang, ingat: **berbagi memori itu cepat tetapi berisiko.**

H.3 Thread di Linux

Linux tidak membedakan proses dan thread secara tajam di dalam kernel. Keduanya adalah *task*, dibuat dengan system call `clone()`, dan yang membedakan hanyalah **seberapa banyak sumber daya yang dibagi**. Jika semuanya dibagi (memori, berkas terbuka, dan lain-lain), hasilnya terlihat sebagai thread. Perintah `ps -elf` menampilkan setiap thread (kolom **LWP**, *light-weight process*) dan **NLWP** menunjukkan jumlah thread tiap proses.

I. MODEL MULTITHREADING

Thread dapat dikelola di dua tingkat. **User thread** dikelola oleh pustaka di user mode tanpa campur tangan kernel. **Kernel thread** dikelola langsung oleh kernel, dan hanya kernel thread yang dijadwalkan CPU. Hubungan keduanya menghasilkan tiga model (Gambar 3.6).



Gambar 3.6 Tiga model multithreading

Model	Kelebihan	Kekurangan
Many-to-one	Pembuatan dan pengelolaan thread sangat murah (di user mode)	Satu <i>blocking call</i> menghentikan semua thread; tidak dapat paralel di multicore
One-to-one	Thread lain tetap jalan saat satu thread menunggu; dapat paralel di multicore	Membuat thread berarti membuat kernel thread, yang lebih mahal; jumlah thread dibatasi
Many-to-many	Banyak user thread dipetakan ke beberapa kernel thread; fleksibel dan murah	Penjadwalan lebih rumit untuk dirancang

Dalam praktik: Linux dan Windows memakai model **one-to-one**. Model many-to-many dipakai oleh *runtime* bahasa tertentu, misalnya *goroutine* di Go, yang dikelola penjadwal runtime Go di atas sejumlah thread OS. Java 21 juga menyediakan *virtual thread* yang bekerja dengan prinsip serupa.

Konteks SI: thread dalam Python

Pada implementasi standar Python (CPython), terdapat *Global Interpreter Lock* (GIL) yang membuat hanya satu thread yang mengeksekusi bytecode Python pada satu waktu. Akibatnya, thread Python bermanfaat untuk tugas yang banyak **menunggu** (I/O seperti jaringan atau basis data), tetapi **tidak mempercepat** tugas yang berat **menghitung** (CPU-bound). Untuk yang kedua, gunakan banyak **proses** (modul **multiprocessing**). Ini contoh nyata bahwa pilihan antara proses dan thread memengaruhi kinerja aplikasi yang Anda bangun. Versi Python terbaru menyediakan build opsional tanpa GIL, tetapi perilaku bawaannya tetap perlu diperiksa pada versi yang dipakai.

J. STUDI KASUS (PROBLEM-BASED LEARNING): PORTAL KRS LAMBAT

Kasus ini fiktif dan dipakai untuk diskusi kelompok 10 menit pada pertemuan tatap muka.

Situasi. Pukul 08.05 pada hari pertama pengisian KRS, mahasiswa Universitas Nusantara mengeluh portal KRS sangat lambat dan beberapa halaman gagal dimuat. Anda adalah staf TI yang login ke server Ubuntu (4 inti CPU, 8 GB RAM) lewat SSH, lalu menjalankan dua perintah. Berikut keluarannya (disederhanakan).

```
$ uptime
08:07:41 up 21 days,  load average: 6.42, 5.10, 3.28

$ ps -eo pid,ppid,stat,ni,pcpu,pmem,etime,comm --sort=-pcpu | head -9
PID  PPID  STAT  NI  %CPU  %MEM  ELAPSED  COMMAND
3318  3302  R      0  98.1   3.9    12:40   python3
3290   1    D      10   3.8   0.1    12:55   tar
1895  1890  Dl      0   1.1   4.7    5-02:11  gunicorn
1896  1890  Dl      0   0.9   4.6    5-02:11  gunicorn
1897  1890  Sl      0   0.4   4.5    5-02:11  gunicorn
1890   1    Ss      0   0.1   1.2    5-02:11  gunicorn
3302   1    S      0   0.0   0.2    13:02   bash
3340  3302  Z      0   0.0   0.0    12:30   python3 <defunct>
```

Keterangan tambahan: **gunicorn** adalah server aplikasi KRS (satu proses induk dan beberapa worker). **tar** adalah pekerjaan backup yang dijalankan oleh timer systemd dengan **nice 10**. **bash** (PID 3302) adalah skrip **rekap.sh** yang dijalankan staf akademik 13 menit lalu dan memulai dua proses **python3**. Pada **top** terlihat persentase **wait** I/O CPU sekitar 25%.

Pertanyaan diskusi:

1. Proses mana yang paling banyak memakai CPU, dan dari mana asalnya? State apa yang ditampilkan dan apa artinya?
2. Load average 6,42 pada server 4 inti. Apakah itu berarti semua inti sibuk? Hubungkan dengan proses ber-state **D** dan nilai **wait** I/O.
3. Apa arti proses **python3 <defunct>** (PID 3340)? Mengapa **kill -9 3340** tidak akan berhasil, dan bagaimana cara menyelesaikannya?
4. Susun **urutan tindakan yang aman** untuk memulihkan kinerja portal tanpa memutus mahasiswa yang sedang mengisi KRS. Mengapa menghentikan **gunicorn** (PID 1890) bukan langkah pertama? Mana yang boleh dilakukan lebih dulu: menurunkan prioritas atau menghentikan proses?
5. Apa yang harus dibenahi agar kejadian ini tidak terulang? (Petunjuk: jadwal backup, batas sumber daya, pemisahan beban rekam dari server utama.)

K. RANGKUMAN

- **Proses** adalah program yang sedang dieksekusi; memiliki ruang memori (text, data, heap, stack), PID, dan konteks eksekusi. Satu program dapat menjadi banyak proses.
- Lima state proses: **new**, **ready**, **running**, **waiting**, **terminated**. Pada Linux terlihat sebagai R, S, D, T, Z pada kolom STAT **ps**.
- **PCB** menyimpan seluruh konteks proses (state, PC, register, info penjadwalan, memori, berkas). Di Linux berupa **task_struct**.

- **Context switch** menyimpan konteks proses lama dan memuat konteks proses baru; ini **overhead** karena CPU tidak mengerjakan pekerjaan berguna saat itu.
- Proses dibuat dengan **fork** (menggandakan), diganti isinya dengan **exec**, ditunggu dengan **wait**, dan diakhiri dengan **exit** atau sinyal. Zombie = selesai tetapi belum di-wait induknya.
- **IPC** ada dua dasar: **shared memory** (cepat, perlu sinkronisasi) dan **message passing** (lebih lambat, lebih mudah dan aman), misalnya pipe dan socket.
- **Thread** berbagi kode, data, heap, dan berkas terbuka tetapi punya register, PC, dan stack sendiri. Lebih ringan dari proses, tetapi crash satu thread dapat menjatuhkan seluruh proses.
- Model multithreading: **many-to-one**, **one-to-one** (Linux, Windows), dan **many-to-many** (runtime tertentu seperti Go).

L. GLOSARIUM

Istilah	Arti Singkat
Proses	Program yang sedang dieksekusi beserta konteksnya
PID / PPID	<i>Process ID</i> dan PID induknya
PCB	<i>Process Control Block</i> , struktur data kernel yang menyimpan semua informasi sebuah proses
State	Keadaan proses: new, ready, running, waiting, terminated
Context switch	Perpindahan CPU dari satu proses ke proses lain dengan menyimpan dan memuat konteks
Preemption	Kernel mengambil alih CPU dari proses yang sedang berjalan, misalnya karena jatah waktu habis
fork / exec / wait	System call untuk menggandakan proses, mengganti programnya, dan menunggu anak selesai
Zombie	Proses yang sudah selesai tetapi statusnya belum dibaca induknya
Orphan	Proses yang induknya selesai lebih dulu, lalu diadopsi PID 1
Sinyal	Notifikasi singkat yang dikirim ke proses (SIGTERM, SIGKILL, dan lain-lain)
IPC	<i>Inter-Process Communication</i> , mekanisme pertukaran data antarproses
Pipe	Saluran satu arah di kernel yang menghubungkan keluaran satu proses ke masukan proses lain
Thread	Satuan eksekusi di dalam proses; berbagi memori dengan thread lain di proses yang sama
Multithreading	Satu proses yang memiliki lebih dari satu thread
Konkurensi vs paralelisme	Bergantian berkemajuan vs benar-benar berjalan serentak
nice	Nilai kebaikan proses (-20 sampai 19); makin besar makin "mengalah" kepada proses lain
Daemon / service	Proses latar belakang yang menyediakan layanan, dikelola oleh systemd
Unit systemd	Berkas konfigurasi yang mendefinisikan sebuah service (<i>.service</i>)

N. LATIHAN MANDIRI

1. Gambarkan diagram state proses dan beri label peristiwa pada setiap panah. Jelaskan mengapa tidak ada panah langsung dari *waiting* ke *running*.
2. Sebutkan minimal lima informasi yang disimpan di PCB. Mengapa PCB mutlak diperlukan agar context switch dapat dilakukan?
3. Jelaskan langkah-langkah context switch dari P0 ke P1. Mengapa jatah waktu (*time slice*) yang terlalu pendek merugikan kinerja?
4. Sebuah program memanggil `fork()` satu kali. Berapa proses yang ada setelahnya? Apa nilai yang diterima masing-masing proses, dan bagaimana program membedakan keduanya?
5. Bandingkan shared memory dan message passing. Pilih satu untuk komunikasi antara dua proses di komputer berbeda dan jelaskan alasannya.
6. Server Anda membuat satu thread per permintaan dan salah satu thread mengalami crash karena bug. Apa dampaknya jika server itu memakai thread, dan apa dampaknya jika memakai proses terpisah?

O. PENUGASAN TERSTRUKTUR: IDENTIFIKASI PROSES DAN SERVICE SERVER APLIKASI

Bentuk: individu. **Batas pengumpulan:** sebelum Pertemuan 4, melalui LMS (PDF, maksimal 3 halaman di luar lampiran tangkapan layar). **Bobot:** bagian dari komponen Latihan Kasus & Laporan Praktikum.

Skenario: Anda adalah administrator sistem baru di sebuah perusahaan. Anda diberi akses SSH ke sebuah server aplikasi web (gunakan VM Ubuntu Server Anda yang telah dipasang satu aplikasi web sederhana, misalnya `python3 -m http.server`, Nginx, atau Apache) dan diminta membuat **inventaris proses dan service** sebagai dokumen serah terima.

Tugas:

1. Jalankan satu aplikasi web (service) di VM Anda dan pastikan ia dikelola oleh `systemd` (boleh memakai unit buatan sendiri dari Praktikum 3 atau memasang paket seperti Nginx).
2. Dengan `ps`, `pstree`, dan `systemctl`, identifikasi **semua proses milik service itu**: PID, PPID, pemilik, state (STAT), jumlah thread, serta pemakaian CPU dan memori. Sajikan dalam tabel.
3. Gambarkan **pohon proses** service tersebut (dari PID 1) dan jelaskan hubungan induk-anaknya.
4. Identifikasi **5 service lain yang berjalan** di VM, jelaskan fungsi masing-masing dan apakah aktif otomatis saat boot (`systemctl is-enabled`).
5. Simulasikan gangguan: hentikan proses utama dengan `kill -9`. Catat apa yang terjadi (apakah dipulihkan otomatis?), lalu jelaskan mengapa, berdasarkan konsep sinyal dan konfigurasi service.
6. Tulis **rekomendasi** (3–5 poin) untuk pengelolaan service ini: pembatasan sumber daya, pemulihan otomatis, pencatatan log, dan hal lain yang Anda anggap penting. Cantumkan daftar pustaka. Jika memakai AI generatif, sebutkan alat dan bagian yang dibantu.

Aspek Penilaian	Bobot
Ketepatan dan kelengkapan inventaris proses dan service (tabel, tangkapan layar)	30%
Ketepatan pohon proses dan penjelasan hubungan induk-anak	20%
Ketepatan analisis simulasi gangguan (sinyal, perilaku service)	25%

Aspek Penilaian	Bobot
Kualitas rekomendasi (relevan dan dapat dijalankan)	15%
Kerapian penyajian dan sitasi	10%

P. PANDUAN PRAKTIKUM 3: ps, top/htop, kill & SIGNAL, nice, DAN systemctl

P.1 Tujuan

1. Mengamati proses, PID, PPID, dan state pada sistem nyata dengan **ps**, **pstree**, dan **/proc**.
2. Memantau pemakaian CPU dan memori dengan **top** dan **htop**, serta membaca *load average*.
3. Mengendalikan proses: latar depan dan latar belakang, mengirim sinyal dengan **kill**, dan menangkap sinyal dalam skrip.
4. Mengamati proses zombie dan context switch.
5. Mengubah prioritas proses dengan **nice** dan **renice**.
6. Membuat dan mengelola service dengan **systemctl** serta membaca lognya dengan **journalctl**.

Persiapan: hubungkan ke VM lewat SSH (Praktikum 2). Seluruh perintah dijalankan di VM. Ubah prompt agar menampilkan NIM Anda (seperti Praktikum 1) sehingga tangkapan layar dapat diverifikasi.

P.2 Langkah 1 — Mengamati Proses dengan ps dan pstree

```
echo $$          # PID shell Anda saat ini
ps              # proses di terminal ini
ps aux | head -15 # semua proses, format pengguna
ps -eo pid,ppid,stat,ni,pcpu,pmem,etime,comm | head -20
pstree -p | head -30 # pohon proses; perhatikan PID 1
ps -elf | head -10  # tampilkan thread (kolom LWP, NLWP)
ps -elf | wc -l     # jumlah seluruh thread
ps -e | wc -l      # jumlah proses
```

Amati dan catat: (a) PID dan PPID shell Anda; (b) siapa induk proses **sshd** dan PID berapa proses pertama; (c) huruf STAT yang paling sering muncul; (d) mana yang lebih banyak, jumlah proses atau jumlah thread, dan mengapa.

Sekarang intip PCB versi Linux. Berkas virtual **/proc/<PID>/** berisi informasi proses yang dibaca langsung dari kernel.

```
head -12 /proc/$$/status # nama, state, PID, PPID, jumlah thread
ls /proc/$$/fd           # file descriptor yang terbuka (0,1,2 = stdin/stdout/stderr)
cat /proc/$$/maps | head -8 # peta memori proses (text, heap, stack)
```

P.3 Langkah 2 — Memantau dengan top dan htop

```
sudo apt update
sudo apt install -y htop
top          # tekan q untuk keluar
htop        # F10 atau q untuk keluar
top -b -n 1 | head -15 # satu kali tampil, untuk disalin ke laporan
```

Di **top**, tekan **P** untuk mengurutkan berdasarkan CPU, **M** berdasarkan memori, dan **1** untuk menampilkan tiap inti CPU. Perhatikan baris ringkasan di bagian atas:

- **load average** (tiga angka: 1, 5, 15 menit terakhir) adalah rata-rata jumlah proses yang sedang berjalan, siap berjalan, atau menunggu I/O tak-terinterupsi (state D). Pada Linux, proses state D ikut dihitung. Bandingkan dengan jumlah inti CPU Anda (**nproc**): load yang konsisten di atas jumlah inti menandakan antrean.
- **%Cpu(s)**: **us** (user), **sy** (kernel), **id** (idle), **wa** (menunggu I/O).
- **Tasks**: jumlah proses total, running, sleeping, stopped, dan zombie.

P.4 Langkah 3 — Latar Belakang, Sinyal, dan kill

```
sleep 300 &          # & = jalankan di latar belakang
jobs                # daftar job di shell ini
sleep 400           # lalu tekan Ctrl+Z (SIGTSTP): proses berhenti
jobs               # lihat state Stopped
bg %2              # lanjutkan di latar belakang (kirim SIGCONT)
ps -o pid,ppid,stat,cmd -C sleep
kill -1            # daftar seluruh sinyal
pkill sleep        # kirim SIGTERM ke semua proses bernama sleep
```

Sekarang buat skrip yang **menangkap** sinyal, untuk membuktikan perbedaan SIGTERM dan SIGKILL. Simpan sebagai **sinyal.sh**:

```
#!/bin/bash
trap 'echo "Menerima SIGTERM, merapikan lalu keluar"; exit 0' TERM
trap 'echo "Ctrl+C ditangkap, saya tidak berhenti"' INT
echo "PID saya: $$"
while true; do sleep 1; done
```

```
chmod +x sinyal.sh
./sinyal.sh &          # catat PID-nya, misal 4321
kill -INT 4321         # ditangkap, skrip tetap hidup
kill -TERM 4321        # ditangkap, skrip merapikan lalu keluar

./sinyal.sh &
kill -9 4321           # SIGKILL tidak dapat ditangkap; tidak ada pesan apa pun
```

Catatan: Bash menjalankan *trap* setelah perintah **sleep 1** yang sedang berjalan selesai, jadi pesan dapat muncul dengan jeda hingga satu detik. **Amati**: pesan mana yang muncul untuk setiap sinyal, dan mengapa **kill -9** tidak memunculkan pesan.

P.5 Langkah 4 — Mengamati Zombie dan Context Switch

Zombie. Simpan program Python berikut sebagai **zombie.py**. Anak berakhir seketika, tetapi induk tidur 60 detik tanpa memanggil **wait()**.

```
import os, time
pid = os.fork()
if pid == 0:
    os._exit(0)          # anak langsung selesai
else:
    print("PID anak:", pid)
    time.sleep(60)       # induk tidak memanggil wait()
```

```
python3 zombie.py &
ps -eo pid,ppid,stat,comm | awk '$3 ~ /Z/'      # tampilkan hanya proses ber-state Z
# tunggu 60 detik sampai induk selesai, lalu ulangi perintah ps; apa yang terjadi?
```

Context switch. Bandingkan proses yang menunggu dengan proses yang terus menghitung.

```
sleep 120 &
grep ctxt /proc/$!/status      # $! = PID job terakhir
yes > /dev/null &
sleep 5; grep ctxt /proc/$!/status
vmstat 1 3                     # kolom cs = jumlah context switch per detik
pkill yes; pkill sleep         # bersihkan
```

Amati: `voluntary_ctxt_switches` dan `nonvoluntary_ctxt_switches` pada kedua proses. Mana yang didominasi *voluntary* dan mana yang *nonvoluntary*? Jelaskan hubungannya dengan state Waiting dan Running.

P.6 Langkah 5 — Prioritas dengan nice dan renice

Nilai *nice* berkisar dari -20 (paling diutamakan) hingga 19 (paling mengalah); bawaannya 0. Pengguna biasa umumnya hanya dapat **menaikkan** nilai nice proses miliknya sendiri (menjadi lebih "baik hati"); menurunkannya memerlukan `sudo`. Agar pengaruhnya terlihat jelas, kita paksa dua proses berebut **satu inti CPU yang sama** dengan `taskset`.

```
taskset -c 0 yes > /dev/null &      # nice 0, di inti 0
taskset -c 0 nice -n 19 yes > /dev/null &  # nice 19, di inti 0
top -b -n 1 -o %CPU | head -12      # bandingkan %CPU dan kolom NI
renice -n 10 -p <PID_nice19>        # ubah nice proses yang sudah berjalan
pkill yes                           # bersihkan
```

Amati: proses ber-nice 0 mendapat bagian CPU yang jauh lebih besar daripada proses ber-nice 19 ketika keduanya berebut inti yang sama. Catat angka %CPU keduanya. Jika tidak berebut inti, perbedaan ini praktis tidak terlihat. Mengapa?

P.7 Langkah 6 — Mengelola Service dengan systemctl

Service (daemon) adalah proses latar belakang yang menyediakan suatu layanan dan dikelola oleh `systemd` (PID 1, Pertemuan 2). Mulailah dengan mengamati service yang sudah ada.

```
systemctl status ssh
systemctl list-units --type=service --state=running
systemctl is-enabled ssh
systemctl cat ssh          # lihat isi unit file-nya
journalctl -u ssh -n 15 --no-pager  # log service ssh
```

Sekarang buat service buatan sendiri: server web sederhana dengan Python.

```
sudo mkdir -p /srv/web
echo "<h1>Halo dari service saya</h1>" | sudo tee /srv/web/index.html
sudo nano /etc/systemd/system/halo.service
```

Isi berkas `halo.service` dengan konfigurasi berikut, lalu simpan.

```
[Unit]
Description=Web sederhana untuk praktikum SO
After=network.target

[Service]
User=nobody
WorkingDirectory=/srv/web
ExecStart=/usr/bin/python3 -m http.server 8080
Restart=on-failure
RestartSec=2

[Install]
WantedBy=multi-user.target
```

```
sudo systemctl daemon-reload          # baca ulang unit file
sudo systemctl start halo
systemctl status halo
curl http://localhost:8080             # jika curl belum ada: sudo apt install -y curl
ps -eo pid,ppid,user,stat,comm | grep http # lihat proses python3 dan induknya (PPID 1)
systemctl show -p MainPID halo         # catat MainPID
sudo kill -9 <MainPID>                 # simulasi crash
sleep 3; systemctl status halo         # apakah hidup lagi? PID-nya sama?
journalctl -u halo -n 20 --no-pager
sudo systemctl enable halo             # aktif otomatis saat boot
systemctl is-enabled halo
sudo systemctl stop halo
sudo systemctl disable halo
```

Amati: (a) siapa induk proses service itu; (b) apa yang dilakukan systemd ketika proses utama dibunuh dengan **kill -9**, dan baris konfigurasi mana yang menyebabkannya; (c) apa perbedaan **start** dengan **enable**.

Awas

Jangan menghentikan service **ssh** atau mengirim **kill -9** ke proses sistem yang tidak Anda kenali, terutama PID 1. Selain memutus koneksi Anda, hal itu dapat membuat VM tidak dapat diakses. Jika terjadi, mulai ulang VM dari VirtualBox.

P.8 Laporan Praktikum 3

Kumpulkan **PDF** melalui LMS sebelum Pertemuan 4 (bobot sesuai skema penilaian pada silabus). Isi laporan:

1. Tangkapan layar (dengan prompt berisi NIM) untuk: **ps tree -p, top -b -n 1 | head -15**, state Stopped dan Zombie, perbandingan context switch, perbandingan nice, serta **systemctl status halo** sebelum dan sesudah **kill -9**.
2. Tabel hasil Langkah 1: PID dan PPID shell, induk **sshd**, jumlah proses dan thread.
3. Jawaban singkat (maksimal 3 kalimat tiap nomor): (1) Mengapa jumlah thread lebih banyak daripada proses? (2) Apa arti load average dan bagaimana membacanya terhadap jumlah inti? (3) Mengapa **kill -9** tidak dapat ditangkap, dan kapan sebaiknya dipakai? (4) Apa hubungan zombie dengan **wait()**? (5) Mengapa nice baru terlihat saat dua proses berebut inti yang sama? (6) Mengapa service **halo** hidup kembali?
4. Refleksi singkat: kendala yang dihadapi dan cara mengatasinya.

P.9 Pemecahan Masalah Umum

Gejala	Penyebab dan solusi
Permission denied saat renice menurunkan nilai	Pengguna biasa hanya boleh menaikkan nilai nice; gunakan sudo untuk menurunkan
Service halo gagal (status failed)	Periksa journalctl -u halo ; cek penulisan unit file, jalur ExecStart , dan pemilik berkas /srv/web
Port 8080 sudah dipakai	Cari prosesnya dengan sudo ss -ltnp grep 8080 , hentikan atau ganti port
Perubahan unit file tidak berpengaruh	Jalankan sudo systemctl daemon-reload lalu restart
yes tidak mau berhenti	pkill yes ; jika gagal, pkill -9 yes
Tidak ada proses zombie terlihat	Jalankan ps dalam 60 detik setelah zombie.py dimulai, sebelum induknya selesai
Perintah tidak ditemukan (htop , taskset)	sudo apt install -y htop util-linux

Q. DAFTAR PUSTAKA

- [1] Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating system concepts* (10th ed.). Wiley.
- [2] Tanenbaum, A. S., & Bos, H. (2023). *Modern operating systems* (5th ed.). Pearson.
- [4] Arpaci-Dusseau, R. H., & Arpaci-Dusseau, A. C. (2018). *Operating systems: Three easy pieces*. Arpaci-Dusseau Books. <https://pages.cs.wisc.edu/~remzi/OSTEP/>
- [5] Nemeth, E., Snyder, G., Hein, T. R., Whaley, B., & Mackin, D. (2017). *UNIX and Linux system administration handbook* (5th ed.). Addison-Wesley.
- [8] Kerrisk, M. (n.d.). *signal(7) — Linux manual page*. <https://man7.org/linux/man-pages/man7/signal.7.html>
- [9] freedesktop.org. (n.d.). *systemd.service — Service unit configuration*. <https://www.freedesktop.org/software/systemd/man/latest/systemd.service.html>

Catatan: nomor [1], [2], [4], dan [5] mengikuti daftar pustaka pada silabus. Nomor [8] dan [9] adalah tambahan khusus pertemuan ini.