Process & OS Structure

Week 2: Proses dan Struktur Sistem Operasi

lectura.id/course/os

Program Studi Teknik Informatika
Institut Teknologi Sumatera

2026

OUTLINE

Peta materi pertemuan ini

1. **Tujuan**
2. **Program dan Proses**
3. **Abstraksi Proses**
4. **Ruang Alamat Proses**
5. **State dan Lifecycle**
6. **Context Switch dan PCB**
7. **PID dan Hirarki Proses**
8. **Dual Mode dan System Call**
9. **Guideline Hands On**
10. **Penutup**

Capaian Pembelajaran

Kemampuan yang ditargetkan setelah kuliah

1. Menjelaskan perbedaan program dan proses secara formal
2. Memahami proses sebagai abstraksi virtualisasi CPU
3. Menjelaskan state proses dan transisinya
4. Mendeskripsikan data penting dalam PCB
5. Memahami batas mode user dan mode kernel
6. Menjelaskan peran `fork`, `exec`, `wait`, `exit`

Target Akhir

Setelah pertemuan ini, mahasiswa mampu membaca alur hidup proses dari dibuat sampai selesai secara **runut**.

Peta Pertemuan Hari Ini

Urutan topik dari dasar ke implementasi

1. Beda formal program dan proses
2. Abstraksi proses pada virtualisasi CPU
3. Model state dan transisinya
1. Struktur data per-proses (PCB)
2. Batas mode user dan mode kernel
3. Peran `fork/exec/wait/exit`

Arah Belajar

Kita bergerak dari konsep **program vs proses**, lalu ke state proses, kemudian ke mode kernel dan system call.

Referensi dari Buku Terjemahan OSTEP

Sumber utama untuk materi Week 2

Sumber Materi

Materi ini merujuk langsung ke pembahasan proses dan API proses pada **folder Book Translate**.

1. `chapter2-book-translate.tex`: proses, state, PCB
2. `chapter3-book-translate.tex`: `fork`, `exec`, `wait`
3. `chapter1-book-translate.tex`: mode user/kernel dan system call

Program vs Proses

Membedakan artefak statis dan entitas berjalan

Analogi cepat: file resep di meja adalah **program**, kegiatan memasak di dapur adalah **proses**.

Program	Proses
Berkas executable di disk	Entitas aktif di CPU/memori
Tidak punya state runtime	Punya state: ready/running/waiting
Belum dijadwalkan OS	Dijadwalkan oleh kernel

Inti Definisi

Program bersifat pasif, sedangkan **proses** bersifat aktif karena sudah mendapat konteks eksekusi.

Satu Program Bisa Jadi Banyak Proses

Dampak ke PID, CPU, dan memori

Ilustrasi: aplikasi browser dibuka 3 jendela. Dari sisi OS, ini bisa jadi beberapa proses berbeda.

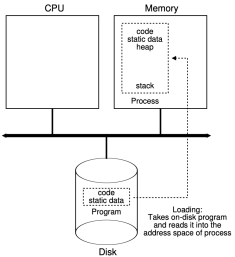
1. Setiap proses punya PID sendiri
2. Setiap proses punya jatah CPU sendiri
3. Setiap proses punya ruang alamat virtual sendiri

Implikasi Penting

Nama aplikasinya sama, tapi setiap proses tetap dicatat terpisah agar kontrol OS tetap **rapi**.

Dari Program Menjadi Proses

Alur pemuatan program ke ruang proses



Ilustrasi OSTEP

OS memuat program dari disk ke memori, lalu mengaktifkannya menjadi proses yang berjalan.

Proses sebagai Abstraksi CPU

Ilusi banyak CPU melalui time-sharing

1. Time-sharing: giliran cepat antar proses
2. Multiprogramming: banyak proses aktif
3. Scheduler memilih proses berikutnya

Proses A $\rightarrow$ Proses B $\rightarrow$ Proses C

Ilusi yang terlihat pengguna: "semua program berjalan bersamaan"

Gagasan Kunci OSTEP

Ilusi ini muncul karena pergantian sangat cepat, bukan karena setiap proses benar-benar punya CPU fisik sendiri.

Kenapa Abstraksi Ini Dibutuhkan?

Isolasi, perlindungan, dan keadilan eksekusi

Tujuan Sistem Operasi

Abstraksi proses membantu OS menjaga **isolasi**, **perlindungan**, dan **keadilan** pemakaian CPU.

1. Isolasi: gangguan proses A tidak langsung menjatuhkan proses B
2. Perlindungan: proses user tidak bisa bebas merusak kernel
3. Keadilan: satu proses tidak memonopoli CPU terus-menerus

Struktur Ruang Alamat Proses

Komponen memori utama milik proses

Komponen Umum

Sebuah proses biasanya punya segmen **code**, **data**, **heap**, dan **stack**.

1. Code/Text: instruksi program
2. Data: variabel global/statis
3. Heap: alokasi dinamis

1. Stack: data pemanggilan fungsi
2. Heap cenderung naik
3. Stack cenderung turun

Dampak Isolasi Ruang Alamat

Kenapa isolasi penting untuk keamanan sistem

Manfaat Praktis

Isolasi ruang alamat membuat sistem lebih **aman** dan lebih **stabil**.

1. Bug memori pada satu proses tidak otomatis merusak proses lain
2. Data proses tidak mudah dibaca proses lain
3. Kernel dapat menghentikan proses bermasalah tanpa mematikan sistem total

Model State Proses

State standar sepanjang siklus hidup proses

Ilustrasi sederhana:

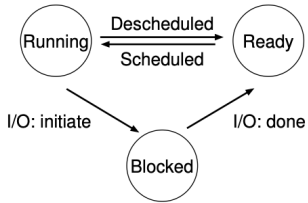
1. **new**: siswa baru masuk kelas
2. **ready**: duduk siap dipanggil
3. **running**: sedang presentasi di depan
4. **waiting**: menunggu giliran alat/proyektor
5. **terminated**: presentasi selesai

Lima State Standar

State proses membantu kernel mengambil keputusan cepat tentang proses mana yang boleh memakai CPU berikutnya.

Diagram State Proses

Visual transisi ready-running-blocked



Cara Baca Diagram

Panah menunjukkan transisi state saat proses dijadwalkan, diblokir karena I/O, lalu siap lagi.

Ilustrasi Antrean CPU

Analogi sederhana pembagian jatah CPU

Bayangkan satu kasir melayani banyak pembeli.

1. Pembeli di baris antrean = state **ready**
2. Pembeli di meja kasir = state **running**
3. Pembeli yang pindah ambil barang = state **waiting**
4. Pembeli keluar toko = state **terminated**

Makna Analogi

CPU itu seperti kasir tunggal; OS bertugas mengatur giliran agar semua proses tetap terlayani.

Tracing Process State: CPU Only

Contoh jejak eksekusi dua proses tanpa I/O

Time	Process0	Process1	Notes
1	Running	Ready	-
2	Running	Ready	-
3	Running	Ready	-
4	Running	Ready	Process0 now done
5	-	Running	-
6	-	Running	-
7	-	Running	-
8	-	Running	Process1 now done

Interpretasi

Selama hanya ada kerja CPU (tanpa I/O), proses bergantian sesuai jatah hingga masing-masing selesai.

Transisi Antar State

Event yang memicu perpindahan state

Transisi	**Pemicu**
ready $\rightarrow$ running	Scheduler memilih proses
running $\rightarrow$ waiting	Proses meminta I/O
running $\rightarrow$ ready	Timer interrupt (preemption)
waiting $\rightarrow$ ready	I/O selesai
running $\rightarrow$ terminated	Proses memanggil `exit`

Pemicu Utama

Perubahan state tidak acak; selalu ada event yang bisa dijelaskan secara sistematis.

Blocking vs Preemption

Dua penyebab proses berhenti memakai CPU

Blocking

1. Umum: menunggu I/O
2. Proses tidak siap jalan
3. Kembali setelah event selesai

Preemption

1. Umum: timer habis
2. Proses dipindah ke ready
3. Tujuan: fairness antar proses

Bedanya Apa?

Blocking berarti proses **menunggu**, sedangkan preemption berarti proses **disela** demi pembagian jatah CPU.

Context Switch: Apa yang Terjadi?

Menyimpan dan memulihkan konteks proses

Ilustrasi: guru memanggil siswa A presentasi, lalu menghentikan sementara, lalu memanggil siswa B.

1. Catatan posisi A harus disimpan dulu
2. B melanjutkan dari catatannya sendiri
3. Setelah itu A bisa dilanjutkan lagi dari titik terakhir

Definisi Konseptual

Context switch adalah simpan konteks proses lama lalu pulihkan konteks proses baru.

Overhead Context Switch

Biaya performa dari pergantian proses

Context switch bukan kerja utama aplikasi, tapi kerja administrasi kernel.

1. Semakin sering switch, semakin besar waktu administrasi
2. Cache CPU bisa kurang efektif setelah pindah proses
3. Sistem harus menyeimbangkan respons dan efisiensi

Fakta Performa

Context switch memang perlu, tetapi terlalu sering dapat menurunkan performa total sistem.

PCB (Process Control Block)

Struktur data inti untuk state proses

Database Mini Tiap Proses

PCB adalah catatan resmi kernel tentang identitas dan kondisi **setiap proses**.

1. PID
2. Process state
3. Snapshot register

1. Metadata penjadwalan
2. Pointer pemetaan memori
3. Referensi file descriptor terbuka

PID, Parent, dan Child

Relasi proses dalam bentuk pohon

Struktur Keluarga Proses

Proses membentuk **pohon proses**; parent membuat child, lalu kernel melacak hubungannya.

1. PID unik untuk identitas
2. Parent-child penting untuk sinkronisasi `wait`
3. Process tree membantu observasi dan pengelolaan sistem

Orphan dan Zombie (Pengantar)

Kondisi khusus pada proses anak

Dua Kondisi yang Sering Ditanya

Orphan: parent selesai duluan. **Zombie**: child selesai, tapi status belum diambil parent.

1. Orphan biasanya diadopsi `init`/`systemd`
2. Zombie menyimpan jejak status proses yang sudah selesai
3. Parent perlu `wait` agar entri zombie dibersihkan

Mode User vs Mode Kernel

Batas hak akses untuk menjaga keamanan OS

Mode User

1. Akses dibatasi
2. Tidak boleh akses device langsung
3. Tidak boleh ubah page table

Mode Kernel

1. Hak akses penuh ke hardware
2. Menjalankan layanan OS
3. Menangani trap/interrupt

Boundary Keamanan

Anggap ini seperti pintu keamanan: user mode ruang publik, kernel mode ruang server inti.

Trap dan Alur System Call

Jalur resmi transisi user ke kernel

Contoh sederhana: aplikasi ingin baca file. Ia tidak bicara langsung ke disk, tapi minta layanan OS.

1. Kode user memanggil system call
2. Hardware memicu trap ke kernel handler
3. Kernel mengeksekusi layanan
4. Return-from-trap mengembalikan kontrol ke mode user

Jalur Resmi User $\rightarrow$ Kernel

Trap membuat perpindahan hak akses menjadi formal, tercatat, dan terkontrol.

Function Call vs System Call

Perbedaan pemanggilan biasa dan layanan kernel

Perbedaan Inti

Function call tetap di ruang user; system call pindah ke kernel untuk layanan yang butuh hak istimewa.

Aspek	**Function Call**	**System Call**
Mode eksekusi	User mode	User $\rightarrow$ Kernel
Hak akses	Biasa	Privileged via kernel
Contoh	Fungsi logika aplikasi	`fork`, `exec`, `wait`, `exit`

Guideline Hands On Kelas

OSTEP Code: folder cpu-api

Repo Hands On: `ostep-code` → folder `cpu-api`

1. Tujuan utama: memahami `fork`, `exec`, `wait`
2. Fokus observasi: perilaku proses di OS nyata
3. Platform: Linux, macOS, atau WSL2 Ubuntu

Target Hands On

Hands On ini menghubungkan teori proses di kelas dengan **perilaku proses yang benar-benar berjalan**.

Prasyarat Minimal

Environment dan tools yang dibutuhkan

OS disarankan

1. Linux (Ubuntu/Debian/Fedora)
2. macOS
3. Windows: gunakan WSL2 Ubuntu

Tools

1. `git`
2. `gcc`/`clang`
3. `make`
4. `ps`, `top`/`htop`, `pstree`

Catatan Penting

Native Windows tanpa WSL tidak cocok untuk **fork()** dan tool POSIX.

Setup Cepat Hands On

Clone dan build source code

1. Clone repo:
2. `git clone https://github.com/remzi-arpacidusseau/ostep-code.git`
3. Masuk folder:
4. `cd ostep-code/cpu-api`
5. Build:
6. `make`

Jika Build Gagal

Compile manual satu file: `gcc -O2 -Wall -Wextra -o demo demo.c`

Menjalankan Program dan Observasi Proses

Langkah run + pemantauan wajib

Menjalankan program

1. `./nama_program`
2. `./nama_program arg1 arg2`
3. Jika perlu: `chmod +x nama_program`

Observasi proses

1. `ps -ef | head`
2. `top` atau `htop`
3. `ps -ef | grep fork`
4. `pstree -p`

Kenapa Ini Wajib

Mahasiswa perlu melihat hubungan **PID, state, dan parent-child** langsung dari sistem.

Kerangka Analisis Kelas

Prediksi, observasi, lalu jelaskan

1. **Prediksi**: API apa yang dipakai? berapa proses muncul?
2. **Observasi**: PID parent-child, urutan output, indikasi zombie/orphan
3. **Jelaskan**: kaitkan ke state proses, context switch, dan efek `exec`/`wait`

Kebiasaan yang Harus Dibangun

Jangan berhenti di output program; selalu jelaskan **mengapa** output tersebut terjadi.

Eksperimen Standar

Tiga eksperimen yang disarankan

1. Ulangi eksekusi 5x untuk melihat nondeterminism
2. Tambahkan `sleep(1)` untuk menonjolkan scheduling
3. Bandingkan versi dengan `wait()` vs tanpa `wait()`

Pertanyaan Inti

Apakah urutan output berubah antar-run, dan apakah `wait()` membuat urutan lebih terkontrol?

Checklist Konsep Setelah Hands On

Wajib nyambung ke materi pertemuan 2

1. Program vs process: beda statis vs dinamis
2. `fork()`: kenapa proses bertambah dan output bisa dobel
3. `exec()`: apa yang diganti dalam address space
4. `wait()`: jaminan sinkronisasi parent-child
5. PID/PPID: representasi relasi proses di OS
6. Nondeterminism: dampak scheduler pada urutan output

Hasil Akhir Hands On

Mahasiswa dapat menghubungkan **API proses** dengan **state proses** secara konkret.

Troubleshooting Cepat

Masalah umum saat setup dan build

1. `make: command not found` → install `make`
2. `gcc: command not found` → install build tools
3. Error header/flag → compile minimal: `gcc -Wall -o x file.c`
4. Pastikan berada di folder `cpu-api`

Catatan Platform

Jika di Windows, gunakan **WSL2 Ubuntu** agar API proses POSIX berjalan sesuai materi.

Ringkasan Konseptual

Poin inti yang harus diingat

Lima Kalimat Kunci

Proses adalah abstraksi utama virtualisasi CPU, dan state proses selalu berada di bawah kontrol kernel.

1. Program statis, proses dinamis
2. Lifecycle proses berubah karena event sistem
3. Context switch bergantung pada data PCB
4. Dual mode menjaga keamanan eksekusi
5. System call adalah pintu resmi interaksi user-kernel

Penutup

Arah belajar lanjutan setelah Week 2

Lanjut Bacaan

Untuk memperkuat konsep, baca ulang bab proses dan API proses di buku terjemahan OSTEP sebelum pertemuan berikutnya.

1. Bab proses: abstraksi, state, PCB
2. Bab API proses: `fork`, `exec`, `wait`
3. Bab dasar OS: system call dan mode eksekusi