

# Operating Systems: Three Easy Pieces

Versi Terjemahan Bahasa Indonesia (Non-Resmi)

Penulis Asli:

**Remzi H. Arpaci-Dusseau and Andrea C.  
Arpaci-Dusseau**

(University of Wisconsin-Madison)

Diterjemahkan Oleh:

(Dengan bantuan AI)

**MCT**

[martin.manullang@if.itera.ac.id](mailto:martin.manullang@if.itera.ac.id)

[mctm.web.id](http://mctm.web.id) | [lectura.id](http://lectura.id)



**Mata Kuliah Sistem Operasi  
IF25-12007**

# **IF ITERA**

## **2026**

---

# PERNYATAAN PENGGUNAAN

## PERNYATAAN PENGGUNAAN

Dengan menggunakan dokumen ini, pengguna menyetujui bahwa:

1. Tidak akan mendistribusikan, mencetak, atau mengedarkan terjemahan ini di luar mata kuliah **IF25-12007 Sistem Operasi**.
2. Terjemahan ini ditujukan semata-mata untuk kepentingan pembelajaran internal dan **tidak resmi**.

### PERINGATAN HUKUM:

Barangsiapa yang mengedarkan dokumen ini secara luas atau menggunakannya untuk tujuan komersial akan berpotensi dituntut secara hukum, karena buku ini **tidak memiliki ijin terjemahan dan ijin edar** dari penerbit atau penulis asli.

*"Gunakanlah dengan bijak dan bertanggung jawab."*



---

# Daftar Isi

<b>1 Pengenalan Sistem Operasi</b>	<b>1</b>
1.1 Virtualisasi CPU . . . . .	2
1.2 Virtualisasi Memori . . . . .	5
1.3 Konkurensi . . . . .	7
1.4 Persistensi . . . . .	9
1.5 Tujuan Desain . . . . .	11
1.6 Sedikit Sejarah . . . . .	12
1.7 Ringkasan . . . . .	16
<b>2 Abstraksi: Proses</b>	<b>19</b>
2.1 Abstraksi: Sebuah Proses . . . . .	20
2.2 API Proses . . . . .	20
2.3 Pembuatan Proses: Lebih Detail . . . . .	20
2.4 State Proses . . . . .	22
2.5 Struktur Data . . . . .	23
2.6 Ringkasan . . . . .	23
<b>3 Antarmuka API Proses</b>	<b>25</b>
3.1 System Call fork() . . . . .	25
3.2 Ruang Alamat Setelah fork() . . . . .	26
3.3 System Call wait() dan waitpid() . . . . .	27
3.4 System Call Keluarga exec() . . . . .	27
3.5 Mengapa API Proses Dirancang Seperti Ini? . . . . .	28
3.6 Redireksi Output dan Pipe . . . . .	28
3.7 Kontrol Proses Dengan Sinyal . . . . .	29
3.8 Pengguna, Hak Akses, dan Superuser . . . . .	29
3.9 Alat Praktis untuk Observasi Proses . . . . .	30
3.10 Ringkasan . . . . .	30
<b>4 Penjadwalan: Pengenalan</b>	<b>31</b>
4.1 Asumsi Beban Kerja . . . . .	31
4.2 Metrik Penjadwalan . . . . .	32

4.3	First In, First Out (FIFO) . . . . .	32
4.4	Shortest Job First (SJF) . . . . .	34
4.5	Shortest Time-to-Completion First (STCF) . . . . .	35
4.6	Metrik Baru: Response Time . . . . .	36
4.7	Round Robin . . . . .	37
4.8	Menggabungkan I/O . . . . .	39
4.9	Tidak Lagi Maha Tahu . . . . .	40
4.10	Ringkasan . . . . .	40
<b>5</b>	<b>Penjadwalan: Multi-Level Feedback Queue</b>	<b>43</b>
5.1	MLFQ: Aturan Dasar . . . . .	44
5.2	Percobaan #1: Cara Mengubah Prioritas . . . . .	45
5.3	Percobaan #2: Priority Boost . . . . .	48
5.4	Percobaan #3: Akuntansi yang Lebih Baik . . . . .	49
5.5	Tuning MLFQ dan Isu Lainnya . . . . .	51
5.6	MLFQ: Ringkasan . . . . .	52

# Pengenalan Sistem Operasi

Jika Anda mengambil mata kuliah sistem operasi tingkat sarjana, Anda seharusnya sudah memiliki gambaran tentang apa yang dilakukan program komputer saat dijalankan. Jika tidak, buku ini (dan mata kuliah yang bersangkutan) akan menjadi sulit — jadi sebaiknya Anda berhenti membaca buku ini, atau lari ke toko buku terdekat dan segera pelajari materi latar belakang yang diperlukan sebelum melanjutkan (buku karya Patt & Patel [PP03] dan Bryant & O'Hallaron [BOH10] adalah buku-buku yang cukup bagus).

Jadi apa yang terjadi ketika sebuah program berjalan?

Yah, program yang berjalan melakukan satu hal yang sangat sederhana: ia mengeksekusi instruksi. Juta-an (dan hari ini, bahkan miliaran) kali setiap detik, prosesor **mengambil** (*fetches*) instruksi dari memori, **mendekode** (*decodes*) instruksi tersebut (yaitu, mencari tahu instruksi apa ini), dan **mengeksekusi-nya** (*executes*) (yaitu, melakukan hal yang seharusnya dilakukan, seperti menjumlahkan dua angka, mengakses memori, memeriksa kondisi, melompat ke fungsi, dan sebagainya). Setelah selesai dengan instruksi ini, prosesor beralih ke instruksi berikutnya, dan seterusnya, dan seterusnya, sampai program akhirnya selesai<sup>1</sup>.

Jadi, kita baru saja menjelaskan dasar-dasar model komputasi **Von Neumann**<sup>2</sup>. Terdengar sederhana, bukan? Tetapi di kelas ini, kita akan belajar bahwa saat program berjalan, banyak hal liar lainnya terjadi dengan tujuan utama membuat sistem **mudah digunakan** (*easy to use*).

Ada sekumpulan perangkat lunak, sebenarnya, yang bertanggung jawab untuk membuatnya mudah menjalankan program (bahkan memungkinkan Anda untuk seolah-olah menjalankan banyak program secara bersamaan), memungkinkan program untuk berbagi memori, memungkinkan program untuk berinteraksi dengan perangkat, dan hal-hal menyenangkan lainnya seperti itu. Kumpulan perangkat lunak itu disebut **sistem operasi (OS)**<sup>3</sup>, karena ia bertugas

---

<sup>1</sup>Tentu saja, prosesor modern melakukan banyak hal aneh dan menakutkan di balik layar untuk membuat program berjalan lebih cepat, mis., mengeksekusi beberapa instruksi sekaligus, dan bahkan mengeluarkan dan menyelesaikannya secara tidak berurutan! Tetapi itu bukan urusan kita di sini; kita hanya peduli dengan model sederhana yang diasumsikan oleh sebagian besar program: bahwa instruksi tampaknya dieksekusi satu per satu, secara teratur dan berurutan.

<sup>2</sup>Von Neumann adalah salah satu pelopor awal sistem komputasi. Dia juga melakukan pekerjaan perintis pada teori permainan (*game theory*) dan bom atom, dan bermain di NBA selama enam tahun. OKE, salah satu dari hal itu tidak benar.

<sup>3</sup>Nama awal lain untuk OS adalah **supervisor** atau bahkan **master control program**. Rupanya, nama

memastikan sistem beroperasi dengan benar dan efisien dalam cara yang mudah digunakan.

## INTI MASALAH: BAGAIMANA MEMVIRTUALISASI SUMBER DAYA

Satu pertanyaan sentral yang akan kita jawab dalam buku ini cukup sederhana: bagaimana sistem operasi memvirtualisasi sumber daya? Ini adalah inti masalah kita. *Mengapa* OS melakukan ini bukanlah pertanyaan utama, karena jawabannya seharusnya jelas: itu membuat sistem lebih mudah digunakan. Jadi, kita fokus pada *bagaimana*: mekanisme dan kebijakan apa yang diimplementasikan oleh OS untuk mencapai virtualisasi? Bagaimana OS melakukannya dengan efisien? Dukungan perangkat keras apa yang diperlukan?

Kami akan menggunakan “inti masalah” (*crux of the problem*), dalam kotak berbayang seperti ini, sebagai cara untuk menyebutkan masalah spesifik yang kami coba selesaikan dalam membangun sistem operasi. Jadi, dalam catatan tentang topik tertentu, Anda mungkin menemukan satu atau lebih inti masalah (ya, ini bentuk jamak yang tepat) yang menyoroti masalahnya. Detail dalam bab ini, tentu saja, menyajikan solusinya, atau setidaknya parameter dasar dari sebuah solusi.

Cara utama OS melakukan ini adalah melalui teknik umum yang kita sebut **virtualisasi**. Yaitu, OS mengambil **sumber daya fisik** (seperti prosesor, atau memori, atau disk) dan mengubahnya menjadi bentuk **virtual** yang lebih umum, kuat, dan mudah digunakan dari dirinya sendiri. Jadi, kita kadang-kadang menyebut sistem operasi sebagai **mesin virtual**.

Tentu saja, agar pengguna dapat memberi tahu OS apa yang harus dilakukan dan dengan demikian memanfaatkan fitur-fitur mesin virtual (seperti menjalankan program, atau mengalokasikan memori, atau mengakses file), OS juga menyediakan beberapa antarmuka (API) yang dapat Anda panggil. Sebuah OS tipikal, pada kenyataannya, mengekspor beberapa ratus **system call** yang tersedia untuk aplikasi. Karena OS menyediakan panggilan-panggilan ini untuk menjalankan program, mengakses memori dan perangkat, dan tindakan terkait lainnya, kita juga kadang-kadang mengatakan bahwa OS menyediakan **pustaka standar** (*standard library*) untuk aplikasi.

Akhirnya, karena virtualisasi memungkinkan banyak program berjalan (sehingga berbagi CPU), dan banyak program untuk secara bersamaan mengakses instruksi dan data mereka sendiri (sehingga berbagi memori), dan banyak program untuk mengakses perangkat (sehingga berbagi disk dan sebagainya), OS kadang-kadang dikenal sebagai **manajer sumber daya**. Setiap CPU, memori, dan disk adalah **sumber daya** sistem; adalah peran sistem operasi untuk **mengelola** sumber daya tersebut, melakukannya secara efisien atau adil atau memang dengan banyak tujuan lain yang mungkin ada dalam pikiran. Untuk memahami peran OS sedikit lebih baik, mari kita lihat beberapa contoh.

### 1.1 Virtualisasi CPU

Gambar 2.1 menggambarkan program pertama kita. Tidak banyak yang dilakukannya. Faktanya, yang dilakukannya hanyalah memanggil `Spin()`, sebuah fungsi yang berulang kali me-

---

yang terakhir terdengar agak terlalu bersemangat (lihat film Tron untuk detailnya) dan untungnya, “sistem operasi” yang lebih populer.

meriksa waktu dan kembali setelah berjalan selama satu detik. Kemudian, ia mencetak string yang diberikan pengguna pada baris perintah, dan mengulanginya, selamanya.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/time.h>
4 #include <assert.h>
5 #include "common.h"

6
7 int
8 main(int argc, char *argv[])
{
9     if (argc != 2) {
10         fprintf(stderr, "usage: cpu <string>\n");
11         exit(1);
12     }
13     char *str = argv[1];
14     while (1) {
15         Spin(1);
16         printf("%s\n", str);
17     }
18     return 0;
}
```

Listing 1.1: Contoh Sederhana: Kode yang Mengulang dan Mencetak (cpu.c)

Mari kita katakan kita menyimpan file ini sebagai `cpu.c` dan memutuskan untuk mengkompilasi dan menjalankannya pada sistem dengan satu prosesor (atau **CPU** sebagaimana kadang-kadang akan kita sebut). Inilah yang akan kita lihat:

```
prompt> gcc -o cpu cpu.c -Wall
prompt> ./cpu "A"
A
A
A
A
^C
prompt>
```

Bukan sebuah proses yang terlalu menarik — sistem mulai menjalankan program, yang berulang kali memeriksa waktu hingga satu detik berlalu. Setelah satu detik berlalu, kode mencetak string input yang diberikan oleh pengguna (dalam contoh ini, huruf “A”), dan berlanjut. Perhatikan program akan berjalan selamanya; dengan menekan “Control-c” (yang pada sistem berbasis UNIX akan menghentikan program yang berjalan di latar depan), kita dapat menghentikan program.

Sekarang, mari kita lakukan hal yang sama, tetapi kali ini, mari kita jalankan banyak contoh berbeda dari program yang sama ini. Gambar 2.2 menunjukkan hasil dari contoh yang sedikit lebih rumit ini.

Yah, sekarang segalanya menjadi sedikit lebih menarik. Meskipun kita hanya memiliki satu prosesor, entah bagaimana keempat program ini tampaknya berjalan pada waktu yang sama!

```
prompt> ./cpu A & ./cpu B & ./cpu C & ./cpu D &
[1] 7353
[2] 7354
[3] 7355
[4] 7356
A
B
D
C
A
B
D
C
A
...
.
```

Gambar 1.1: Menjalankan Banyak Program Sekaligus

Bagaimana keajaiban ini terjadi?<sup>4</sup>

Ternyata sistem operasi, dengan sedikit bantuan dari perangkat keras, bertanggung jawab atas **ilusi** ini, yaitu, ilusi bahwa sistem memiliki sejumlah besar CPU virtual. Mengubah satu CPU (atau sekumpulan kecil CPU) menjadi sejumlah CPU yang tampaknya tak terbatas dan dengan demikian memungkinkan banyak program untuk tampaknya berjalan sekaligus adalah apa yang kita sebut **memvirtualisasi CPU**, fokus dari bagian utama pertama buku ini.

Tentu saja, untuk menjalankan program, dan menghentikannya, dan sebaliknya memberi tahu OS program mana yang akan dijalankan, perlu ada beberapa antarmuka (API) yang dapat Anda gunakan untuk mengomunikasikan keinginan Anda kepada OS. Kita akan berbicara tentang API ini di seluruh buku ini; memang, mereka adalah cara utama sebagian besar pengguna berinteraksi dengan sistem operasi.

Anda mungkin juga memperhatikan bahwa kemampuan untuk menjalankan beberapa program sekaligus memunculkan segala macam pertanyaan baru. Misalnya, jika dua program ingin berjalan pada waktu tertentu, mana yang harus berjalan? Pertanyaan ini dijawab oleh **kebijakan (policy)** OS; kebijakan digunakan di banyak tempat berbeda dalam OS untuk menjawab jenis pertanyaan ini, dan dengan demikian kita akan mempelajarinya saat kita mempelajari tentang **mekanisme** dasar yang diimplementasikan sistem operasi (seperti kemampuan untuk menjalankan beberapa program sekaligus). Karenanya peran OS sebagai **manajer sumber daya**.

---

<sup>4</sup>Perhatikan bagaimana kita menjalankan empat proses pada waktu yang sama, dengan menggunakan simbol `&`. Melakukan hal itu menjalankan pekerjaan di latar belakang (*background*) dalam shell `zsh`, yang berarti pengguna dapat segera mengeluarkan perintah berikutnya, yang dalam kasus ini adalah program lain untuk dijalankan. Jika Anda menggunakan shell yang berbeda (mis., `tcsh`), cara kerjanya sedikit berbeda; baca dokumentasi online untuk detailnya.

## 1.2 Virtualisasi Memori

Sekarang mari kita pertimbangkan memori. Model **memori fisik** yang disajikan oleh mesin modern sangat sederhana. Memori hanyalah sebuah array byte; untuk **membaca** memori, seseorang harus menentukan **alamat** agar dapat mengakses data yang disimpan di sana; untuk **menulis** (atau **memperbarui**) memori, seseorang juga harus menentukan data yang akan ditulis ke alamat yang diberikan.

Memori diakses sepanjang waktu saat program berjalan. Sebuah program menyimpan semua struktur datanya di memori, dan mengaksesnya melalui berbagai instruksi, seperti *loads* dan *stores* atau instruksi eksplisit lainnya yang mengakses memori dalam melakukan pekerjaan mereka. Jangan lupa bahwa setiap instruksi program ada di memori juga; jadi memori diakses pada setiap pengambilan instruksi (*instruction fetch*).

Mari kita lihat sebuah program (pada Gambar 2.3) yang mengalokasikan beberapa memori dengan memanggil `malloc()`. Output dari program ini dapat ditemukan di sini:

```
1 #include <unistd.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include "common.h"

5
6 int
7 main(int argc, char *argv[])
{
8
9     int *p = malloc(sizeof(int)); // a1
10    assert(p != NULL);
11    printf("(2134) address pointed to by p: %p\n",
12           getpid(), p); // a2
13    *p = 0; // a3
14    while (1) {
15        Spin(1);
16        *p = *p + 1;
17        printf("(2134) p: %d\n", getpid(), *p); // a4
18    }
19    return 0;
20 }
```

Listing 1.2: Program yang Mengakses Memori (mem.c)

```
prompt> ./mem
(2134) address pointed to by p: 0x200000
(2134) p: 1
(2134) p: 2
(2134) p: 3
(2134) p: 4
(2134) p: 5
^C
```

Program ini melakukan beberapa hal. Pertama, ia mengalokasikan beberapa memori (baris a1). Kemudian, ia mencetak alamat memori (a2), dan kemudian menempatkan angka nol ke dalam slot pertama dari memori yang baru dialokasikan (a3). Akhirnya, ia melakukan loop,

menunda selama satu detik dan menaikkan nilai yang disimpan di alamat yang dipegang dalam p. Dengan setiap pernyataan cetak, ia juga mencetak apa yang disebut pengidentifikasi proses (*Process Identifier* atau PID) dari program yang sedang berjalan. PID ini unik per proses yang berjalan.

Sekali lagi, hasil pertama ini tidak terlalu menarik. Memori yang baru dialokasikan berada di alamat 0x200000. Saat program berjalan, perlahan-lahan ia memperbarui nilai dan mencetak hasilnya.

Sekarang, kita menjalankan lagi beberapa contoh dari program yang sama ini untuk melihat apa yang terjadi (Gambar 2.4).

```
prompt> ./mem & ./mem &
[1] 24113
[2] 24114
(24113) address pointed to by p: 0x200000
(24114) address pointed to by p: 0x200000
(24113) p: 1
(24114) p: 1
(24114) p: 2
(24113) p: 2
(24113) p: 3
(24114) p: 3
(24113) p: 4
(24114) p: 4
...
...
```

Gambar 1.2: Menjalankan Program Memori Beberapa Kali

Kita melihat dari contoh bahwa setiap program yang berjalan telah mengalokasikan memori pada alamat yang sama (0x200000), namun masing-masing tampaknya memperbarui nilai pada 0x200000 secara independen! Seolah-olah setiap program yang berjalan memiliki memori pribadinya sendiri, alih-alih berbagi memori fisik yang sama dengan program yang berjalan lainnya<sup>5</sup>.

Memang, itulah tepatnya yang terjadi di sini karena OS sedang **memvirtualisasi memori**. Setiap proses mengakses **ruang alamat virtual** pribadinya sendiri (kadang-kadang hanya disebut **ruang alamat**), yang entah bagaimana dipetakan oleh OS ke memori fisik mesin. Referensi memori dalam satu program yang berjalan tidak mempengaruhi ruang alamat proses lain (atau OS itu sendiri); sejauh menyangkut program yang sedang berjalan, ia memiliki memori fisik untuk dirinya sendiri. Kenyataannya, bagaimanapun, adalah bahwa memori fisik adalah sumber daya bersama, dikelola oleh sistem operasi. Tepatnya bagaimana semua ini diselesaikan juga merupakan subjek dari bagian pertama buku ini, tentang topik **virtualisasi**.

---

<sup>5</sup>Agar contoh ini berfungsi, Anda perlu memastikan pengacakan ruang alamat (*address-space randomization*) dinonaktifkan; pengacak, ternyata, bisa menjadi pertahanan yang baik terhadap jenis cacat keamanan tertentu. Baca lebih lanjut tentang itu sendiri, terutama jika Anda ingin belajar cara membobol sistem komputer melalui serangan *stack-smashing*. Bukan berarti kami merekomendasikan hal seperti itu...

## 1.3 Konkurensi

Tema utama lain dari buku ini adalah **konkurensi**. Kami menggunakan istilah konseptual ini untuk merujuk pada sejumlah masalah yang muncul, dan harus ditangani, ketika mengerjakan banyak hal sekaligus (yani, secara konkuren) dalam program yang sama. Masalah konkurensi muncul pertama kali di dalam sistem operasi itu sendiri; seperti yang dapat Anda lihat dalam contoh-contoh di atas tentang virtualisasi, OS menyulap banyak hal sekaligus, pertama menjalankan satu proses, lalu yang lain, dan seterusnya. Ternyata, melakukan hal itu mengarah pada beberapa masalah yang mendalam dan menarik.

Sayangnya, masalah konkurensi tidak lagi terbatas hanya pada OS itu sendiri. Memang, program **multi-threaded** modern menunjukkan masalah yang sama. Mari kita tunjukkan dengan contoh program multi-threaded (Gambar 2.5).

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "common.h"
4 #include "common_threads.h"

5
6 volatile int counter = 0;
7 int loops;

8
9 void *worker(void *arg) {
10     int i;
11     for (i = 0; i < loops; i++) {
12         counter++;
13     }
14     return NULL;
15 }

16
17 int main(int argc, char *argv[]) {
18     if (argc != 2) {
19         fprintf(stderr, "usage: threads <value>\n");
20         exit(1);
21     }
22     loops = atoi(argv[1]);
23     pthread_t p1, p2;
24     printf("Initial value : %d\n", counter);

25     Pthread_create(&p1, NULL, worker, NULL);
26     Pthread_create(&p2, NULL, worker, NULL);
27     Pthread_join(p1, NULL);
28     Pthread_join(p2, NULL);
29     printf("Final value : %d\n", counter);
30     return 0;
31 }
```

Listing 1.3: Program Multi-threaded (*threads.c*)

Meskipun Anda mungkin tidak memahami contoh ini sepenuhnya saat ini (dan kita akan mempelajari lebih lanjut tentangnya di bab-bab selanjutnya, di bagian buku tentang konkurensi), ide dasarnya sederhana. Program utama membuat dua **utas** (*threads*) menggunakan

`Pthread_create()`<sup>6</sup>. Anda dapat memikirkan utas sebagai fungsi yang berjalan dalam ruang memori yang sama dengan fungsi lain, dengan lebih dari satu aktif pada satu waktu. Dalam contoh ini, setiap utas mulai berjalan dalam rutinitas yang disebut `worker()`, di mana ia hanya menaikkan penghitung (*counter*) dalam satu loop sebanyak `loops` kali.

Di bawah ini adalah transkrip tentang apa yang terjadi ketika kita menjalankan program ini dengan nilai input untuk variabel `loops` diatur ke 1000. Nilai `loops` menentukan berapa kali masing-masing dari dua pekerja akan menaikkan penghitung bersama dalam satu loop. Ketika program dijalankan dengan nilai `loops` diatur ke 1000, berapa nilai akhir `counter` yang Anda harapkan?

```
prompt> gcc -o threads threads.c -Wall -pthread
prompt> ./threads 1000
Initial value : 0
Final value : 2000
```

Seperti yang mungkin Anda duga, ketika kedua utas selesai, nilai akhir penghitung adalah 2000, karena setiap utas menaikkan penghitung 1000 kali. Memang, ketika nilai input `loops` diatur ke  $N$ , kita akan mengharapkan output akhir dari program menjadi  $2N$ . Tetapi hidup tidak sesederhana itu, ternyata. Mari kita jalankan program yang sama, tetapi dengan nilai yang lebih tinggi untuk `loops`, dan lihat apa yang terjadi:

```
prompt> ./threads 100000
Initial value : 0
Final value : 143012 // huh??
prompt> ./threads 100000
Initial value : 0
Final value : 137298 // apa ini??
```

Dalam proses ini, ketika kita memberikan nilai input 100.000, alih-alih mendapatkan nilai akhir 200.000, kita malah pertama kali mendapatkan 143.012. Kemudian, ketika kita menjalankan program untuk kedua kalinya, kita tidak hanya mendapatkan nilai yang salah lagi, tetapi juga nilai yang *berbeda* dari yang terakhir kali. Faktanya, jika Anda menjalankan program berulang kali dengan nilai `loops` yang tinggi, Anda mungkin menemukan bahwa kadang-kadang Anda bahkan mendapatkan jawaban yang benar! Jadi mengapa ini terjadi?

Ternyata, alasan untuk hasil aneh dan tidak biasa ini berkaitan dengan bagaimana instruksi dieksekusi, yaitu satu per satu. Sayangnya, bagian penting dari program di atas, di mana penghitung bersama dinaikkan, mengambil tiga instruksi: satu untuk memuat nilai penghitung dari memori ke register, satu untuk menaikkannya, dan satu untuk menyimpannya kembali ke memori. Karena ketiga instruksi ini tidak dieksekusi secara **atomik** (sekaligus), hal-hal aneh bisa terjadi. Masalah **konkurensi** inilah yang akan kita bahas secara sangat rinci di bagian kedua buku ini.

---

<sup>6</sup>Panggilan sebenarnya harus ke `pthread_create()` huruf kecil; versi huruf besar adalah pembungkus (*wrapper*) kami sendiri yang memanggil `pthread_create()` dan memastikan bahwa kode pengembalian menunjukkan bahwa panggilan berhasil. Lihat kode untuk detailnya.

## INTI MASALAH: BAGAIMANA MEMBANGUN PROGRAM KONKUREN YANG BENAR

Ketika ada banyak utas yang dieksekusi secara konkuren dalam ruang memori yang sama, bagaimana kita bisa membangun program yang bekerja dengan benar? Primitif apa yang dibutuhkan dari OS? Mekanisme apa yang harus disediakan oleh perangkat keras? Bagaimana kita bisa menggunakan untuk memecahkan masalah konkurensi?

### 1.4 Persistensi

Tema utama ketiga dari kursus ini adalah **persistensi**. Dalam memori sistem, data dapat dengan mudah hilang, karena perangkat seperti DRAM menyimpan nilai-nilai dalam cara yang **volatil** (*volatile*); ketika daya hilang atau sistem macet, data apa pun di memori akan hilang. Oleh karena itu, kita membutuhkan perangkat keras dan perangkat lunak untuk dapat menyimpan data secara **persistent**; penyimpanan seperti itu sangat penting bagi sistem mana pun karena pengguna sangat peduli dengan data mereka.

Perangkat keras datang dalam bentuk semacam perangkat **input/output** atau **I/O**; dalam sistem modern, **hard drive** adalah repositori umum untuk informasi jangka panjang, meskipun **solid-state drive** (SSD) juga membuat kemajuan di arena ini.

Perangkat lunak dalam sistem operasi yang biasanya mengelola disk disebut **sistem file** (*file system*); dengan demikian bertanggung jawab untuk menyimpan **file** apa pun yang dibuat pengguna dengan cara yang andal dan efisien pada disk sistem.

Tidak seperti abstraksi yang disediakan oleh OS untuk CPU dan memori, OS tidak membuat disk virtual pribadi untuk setiap aplikasi. Sebaliknya, diasumsikan bahwa sering kali, pengguna ingin **berbagi** informasi yang ada dalam file. Misalnya, saat menulis program C, Anda mungkin pertama kali menggunakan editor (mis., Emacs<sup>7</sup>) untuk membuat dan mengedit file C (`emacs -nw main.c`). Setelah selesai, Anda dapat menggunakan kompiler untuk mengubah kode sumber menjadi executable (mis., `gcc -o main main.c`). Setelah selesai, Anda dapat menjalankan executable baru (mis., `./main`). Jadi, Anda dapat melihat bagaimana file dibagikan di berbagai proses. Pertama, Emacs membuat file yang berfungsi sebagai input ke kompiler; kompiler menggunakan file input itu untuk membuat file executable baru (dalam banyak langkah — ambil kursus kompiler untuk detailnya); akhirnya, executable baru kemudian dijalankan. Dan dengan demikian program baru lahir!

Untuk memahami ini dengan lebih baik, mari kita lihat beberapa kode. Gambar 2.6 menyajikan kode untuk membuat file (`/tmp/file`) yang berisi string “hello world”.

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <assert.h>
4 #include <fcntl.h>
5 #include <sys/types.h>
```

<sup>7</sup>Anda harus menggunakan Emacs. Jika Anda menggunakan vi, mungkin ada sesuatu yang salah dengan Anda. Jika Anda menggunakan sesuatu yang bukan editor kode asli, itu bahkan lebih buruk.

```

7 int main(int argc, char *argv[]) {
8     int fd = open("/tmp/file",
9                 O_WRONLY|O_CREAT|O_TRUNC,
10                S_IRWXU);
11    assert(fd > -1);
12    int rc = write(fd, "hello world\n", 12);
13    assert(rc == 12);
14    close(fd);
15    return 0;
16 }
```

Listing 1.4: Program yang Melakukan I/O (io.c)

Untuk menyelesaikan tugas ini, program membuat tiga panggilan ke sistem operasi. Yang pertama, panggilan ke `open()`, membuka file dan membuatnya; yang kedua, `write()`, menulis beberapa data ke file; yang ketiga, `close()`, hanya menutup file sehingga menunjukkan program tidak akan menulis data lagi ke dalamnya. **System calls** ini dialihkan ke bagian sistem operasi yang disebut **sistem file**, yang kemudian menangani permintaan dan mengembalikan semacam kode kesalahan kepada pengguna.

Anda mungkin bertanya-tanya apa yang dilakukan OS untuk benar-benar menulis ke disk. Kami akan menunjukkannya kepada Anda tetapi Anda harus berjanji untuk menutup mata Anda terlebih dahulu; itu sangat tidak menyenangkan. Sistem file harus melakukan cukup banyak pekerjaan: pertama mencari tahu di mana di disk data baru ini akan berada, dan kemudian melacaknya dalam berbagai struktur yang dikelola sistem file. Melakukan hal itu memerlukan penerbitan permintaan I/O ke perangkat penyimpanan yang mendasarnya, untuk membaca struktur yang ada atau memperbaruiinya (menulis). Siapa pun yang pernah menulis **device driver**<sup>8</sup> tahu, membuat perangkat melakukan sesuatu atas nama Anda adalah proses yang rumit dan mendetail. Ini membutuhkan pengetahuan yang mendalam tentang antarmuka perangkat tingkat rendah dan semantiknya yang tepat. Untungnya, OS menyediakan cara standar dan sederhana untuk mengakses perangkat melalui panggilan sistemnya. Jadi, OS kadang-kadang dilihat sebagai **pustaka standar**.

Tentu saja, ada lebih banyak detail tentang bagaimana perangkat diakses, dan bagaimana sistem file mengelola data secara persisten di atas perangkat tersebut. Untuk alasan kinerja, sebagian besar sistem file pertama-tama menunda penulisan semacam itu untuk sementara waktu, berharap untuk mengelompokkannya menjadi grup yang lebih besar. Untuk menangani masalah kerusakan sistem selama penulisan, sebagian besar sistem file menggabungkan semacam protokol penulisan yang rumit, seperti **journaling** atau **copy-on-write**, dengan hati-hati memesan penulisan ke disk untuk memastikan bahwa jika kegagalan terjadi selama urutan penulisan, sistem dapat memulihkan keadaan yang wajar sesudahnya. Untuk membuat operasi umum yang berbeda menjadi efisien, sistem file menggunakan banyak struktur data dan metode akses yang berbeda, dari daftar sederhana hingga b-tree yang kompleks. Jika semua ini belum masuk akal, bagus! Kami akan membicarakan semua hal ini lebih banyak di bagian ketiga buku ini tentang **persistensi**, di mana kita akan membahas perangkat dan I/O secara umum, dan kemudian disk, RAID, dan sistem file dengan sangat rinci.

---

<sup>8</sup>Device driver adalah beberapa kode dalam sistem operasi yang tahu cara menangani perangkat tertentu. Kita akan berbicara lebih banyak tentang perangkat dan driver perangkat nanti.

## INTI MASALAH: BAGAIMANA MENYIMPAN DATA SECARA PERSISTEN

Sistem file adalah bagian dari OS yang bertanggung jawab mengelola data persisten. Teknik apa yang diperlukan untuk melakukannya dengan benar? Mekanisme dan kebijakan apa yang diperlukan untuk melakukannya dengan kinerja tinggi? Bagaimana keandalan dicapai, dalam menghadapi kegagalan perangkat keras dan perangkat lunak?

### 1.5 Tujuan Desain

Jadi sekarang Anda memiliki gagasan tentang apa yang sebenarnya dilakukan OS: ia mengambil **sumber daya** fisik, seperti CPU, memori, atau disk, dan **memvirtualisasikannya**. Ia menangani masalah sulit dan rumit terkait dengan **konkurensi**. Dan ia menyimpan file secara **persisten**, sehingga membuatnya aman dalam jangka panjang. Mengingat bahwa kita ingin membangun sistem seperti itu, kita ingin memiliki beberapa tujuan dalam pikiran untuk membantu memfokuskan desain dan implementasi kita dan membuat *trade-off* seperlunya; menemukan set *trade-off* yang tepat adalah kunci untuk membangun sistem.

Salah satu tujuan paling mendasar adalah membangun beberapa **abstraksi** agar sistem nyaman dan mudah digunakan. Abstraksi sangat mendasar untuk semua yang kita lakukan dalam ilmu komputer. Abstraksi memungkinkan untuk menulis program besar dengan membaginya menjadi potongan-potongan kecil dan dapat dimengerti, untuk menulis program seperti itu dalam bahasa tingkat tinggi seperti C<sup>9</sup> tanpa memikirkan tentang assembly, untuk menulis kode dalam assembly tanpa memikirkan tentang gerbang logika, dan untuk membangun prosesor dari gerbang tanpa memikirkan terlalu banyak tentang transistor. Abstraksi sangat mendasar sehingga kadang-kadang kita melupakan pentingnya, tetapi kita tidak akan melupakannya di sini; jadi, di setiap bagian, kita akan membahas beberapa abstraksi utama yang telah berkembang dari waktu ke waktu, memberi Anda cara untuk memikirkan tentang bagian-bagian dari OS.

Satu tujuan dalam merancang dan mengimplementasikan sistem operasi adalah untuk memberikan **kinerja tinggi**; cara lain untuk mengatakan ini adalah tujuan kami adalah untuk **meminimalkan overhead** dari OS. Virtualisasi dan membuat sistem mudah digunakan sangat berharga, tetapi tidak dengan biaya berapa pun; jadi, kita harus berusaha untuk menyediakan virtualisasi dan fitur OS lainnya tanpa overhead yang berlebihan. Overhead ini muncul dalam sejumlah bentuk: waktu ekstra (lebih banyak instruksi) dan ruang ekstra (di memori atau di disk). Kami akan mencari solusi yang meminimalkan satu atau yang lain atau keduanya, jika memungkinkan. Kesempurnaan, bagaimanapun, tidak selalu dapat dicapai, sesuatu yang akan kita pelajari untuk diperhatikan dan (jika perlu) ditoleransi.

Tujuan lain adalah memberikan **perlindungan** (*protection*) antara aplikasi, serta antara OS dan aplikasi. Karena kami ingin mengizinkan banyak program berjalan pada saat yang sama, kami ingin memastikan bahwa perilaku buruk yang berbahaya atau tidak disengaja dari satu program tidak membahayakan yang lain; kita tentu tidak ingin aplikasi dapat membahayakan OS itu sendiri (karena itu akan mempengaruhi semua program yang berjalan pada sistem).

---

<sup>9</sup>Beberapa dari Anda mungkin keberatan menyebut C sebagai bahasa tingkat tinggi. Ingat ini adalah kursus OS, di mana kami senang tidak harus mengkode dalam assembly sepanjang waktu!

Perlindungan adalah inti dari salah satu prinsip utama yang mendasari sistem operasi, yaitu **isolasi**; mengisolasi proses satu sama lain adalah kunci perlindungan dan dengan demikian mendasari banyak hal yang harus dilakukan OS.

Sistem operasi juga harus berjalan tanpa henti; ketika gagal, semua aplikasi yang berjalan pada sistem juga gagal. Karena ketergantungan ini, sistem operasi sering berusaha untuk memberikan tingkat **keandalan** (*reliability*) yang tinggi. Seiring berkembangnya sistem operasi yang semakin kompleks (kadang-kadang berisi jutaan baris kode), membangun sistem operasi yang andal adalah tantangan yang cukup besar — dan memang, banyak penelitian yang sedang berlangsung di lapangan (termasuk beberapa dari pekerjaan kami sendiri [BS+09, SS+10]) berfokus pada masalah yang tepat ini.

Tujuan lain masuk akal: **efisiensi energi** penting di dunia kita yang semakin hijau; **keamanan** (perpanjangan dari perlindungan, sungguh) terhadap aplikasi berbahaya sangat penting, terutama di masa-masa yang sangat terhubung jaringan ini; **mobilitas** semakin penting karena OS dijalankan pada perangkat yang lebih kecil dan lebih kecil. Tergantung pada bagaimana sistem digunakan, OS akan memiliki tujuan yang berbeda dan dengan demikian kemungkinan diimplementasikan setidaknya dengan cara yang sedikit berbeda. Namun, seperti yang akan kita lihat, banyak prinsip yang akan kami sajikan tentang cara membangun OS berguna pada berbagai perangkat yang berbeda.

## 1.6 Sedikit Sejarah

Sebelum menutup pengantar ini, mari kita sajikan sejarah singkat tentang bagaimana sistem operasi berkembang. Seperti sistem apa pun yang dibangun oleh manusia, ide-ide bagus terakumulasi dalam sistem operasi dari waktu ke waktu, ketika para insinyur mempelajari apa yang penting dalam desain mereka. Di sini, kita membahas beberapa perkembangan utama. Untuk pembahasan yang lebih kaya, lihat sejarah sistem operasi yang sangat baik oleh Brinch Hansen [BH00].

### Sistem Operasi Awal: Hanya Pustaka

Pada awalnya, sistem operasi tidak melakukan terlalu banyak. Pada dasarnya, itu hanyalah sekumpulan pustaka dari fungsi yang umum digunakan; misalnya, alih-alih meminta setiap pemrogram sistem menulis kode penanganan I/O tingkat rendah, “OS” akan menyediakan API semacam itu, dan dengan demikian membuat hidup lebih mudah bagi pengembang.

Biasanya, pada sistem mainframe lama ini, satu program berjalan pada satu waktu, dikenalkan oleh operator manusia. Sebagian besar dari apa yang Anda pikir akan dilakukan OS modern (mis., memutuskan urutan menjalankan pekerjaan) dilakukan oleh operator ini. Jika Anda adalah pengembang yang cerdas, Anda akan bersikap baik kepada operator ini, sehingga mereka mungkin memindahkan pekerjaan Anda ke bagian depan antrian.

Mode komputasi ini dikenal sebagai **pemrosesan batch** (*batch processing*), karena sejumlah pekerjaan disiapkan dan kemudian dijalankan dalam “batch” oleh operator. Komputer, pada saat itu, tidak digunakan secara interaktif, karena biaya: itu terlalu mahal untuk membiarkan pengguna duduk di depan komputer dan menggunakanannya, karena sebagian besar waktu itu hanya akan duduk diam, menghabiskan biaya fasilitas ratusan ribu dolar per jam [BH00].

## Melampaui Pustaka: Perlindungan

Dalam bergerak melampaui sekadar pustaka layanan yang umum digunakan, sistem operasi mengambil peran yang lebih sentral dalam mengelola mesin. Satu aspek penting dari ini adalah kesadaran bahwa kode yang dijalankan atas nama OS adalah istimewa; ia memiliki kendali atas perangkat dan dengan demikian harus diperlakukan secara berbeda dari kode aplikasi normal. Mengapa ini? Yah, bayangkan jika Anda mengizinkan aplikasi apa pun untuk membaca dari mana saja di disk; gagasan privasi keluar jendela, karena program apa pun dapat membaca file apa pun. Jadi, menerapkan **sistem file** (untuk mengelola file Anda) sebagai pustaka tidak masuk akal. Sebaliknya, sesuatu yang lain diperlukan.

Dengan demikian, gagasan tentang **system call** ditemukan, dipelopori oleh sistem komputasi Atlas [K+61,L78]. Alih-alih menyediakan rutinitas OS sebagai pustaka (di mana Anda baru saja membuat **panggilan prosedur** untuk mengaksesnya), idenya di sini adalah menambahkan sepasang instruksi perangkat keras khusus dan status perangkat keras untuk membuat transisi ke OS menjadi proses yang lebih formal dan terkontrol.

Perbedaan utama antara panggilan sistem dan panggilan prosedur adalah bahwa panggilan sistem mentransfer kontrol (yaitu, melompat) ke OS sambil secara bersamaan menaikkan **tingkat hak istimewa perangkat keras** (*hardware privilege level*). Aplikasi pengguna berjalan dalam apa yang disebut sebagai **mode pengguna** (*user mode*) yang berarti perangkat keras membatasi apa yang dapat dilakukan aplikasi; misalnya, aplikasi yang berjalan dalam mode pengguna biasanya tidak dapat memulai permintaan I/O ke disk, mengakses halaman memori fisik apa pun, atau mengirim paket di jaringan. Ketika panggilan sistem dimulai (biasanya melalui instruksi perangkat keras khusus yang disebut **trap**), perangkat keras mentransfer kontrol ke **trap handler** yang telah ditentukan sebelumnya (yang disiapkan OS sebelumnya) dan secara bersamaan menaikkan tingkat hak istimewa ke **mode kernel**. Dalam mode kernel, OS memiliki akses penuh ke perangkat keras sistem dan dengan demikian dapat melakukan hal-hal seperti memulai permintaan I/O atau membuat lebih banyak memori tersedia untuk program. Ketika OS selesai melayani permintaan, ia mengembalikan kontrol kembali ke pengguna melalui instruksi **return-from-trap** khusus, yang kembali ke mode pengguna sambil secara bersamaan mengembalikan kontrol kembali ke tempat aplikasi tinggalkan.

## INFO TAMBAHAN: PENTINGNYA UNIX

Sulit untuk melebih-lebihkan pentingnya UNIX dalam sejarah sistem operasi. Dipeengaruhi oleh sistem sebelumnya (khususnya, sistem **Multics** yang terkenal dari MIT), UNIX menyatukan banyak ide hebat dan membuat sistem yang sederhana namun kuat. Mendasari UNIX asli “Bell Labs” adalah prinsip pemersatu membangun program kecil yang kuat yang dapat dihubungkan bersama untuk membentuk alur kerja yang lebih besar. **Shell**, tempat Anda mengetik perintah, menyediakan primitif seperti **pipa** (*pipes*) untuk memungkinkan pemrograman tingkat meta seperti itu, dan dengan demikian menjadi mudah untuk merangkai program bersama untuk menyelesaikan tugas yang lebih besar. Misalnya, untuk menemukan baris file teks yang memiliki kata “foo” di dalamnya, dan kemudian menghitung berapa banyak baris seperti itu ada, Anda akan mengetik: `grep foo file.txt|wc -l`, dengan demikian menggunakan program `grep` dan `wc` (hitung kata) untuk mencapai tugas Anda.

Lingkungan UNIX ramah bagi pemrogram dan pengembang, juga menyediakan kompiler untuk **bahasa pemrograman C** yang baru. Memudahkannya bagi pemrogram untuk menulis program mereka sendiri, serta membagikannya, membuat UNIX sangat populer. Dan mungkin sangat membantu bahwa penulis memberikan salinan secara gratis kepada siapa saja yang meminta, bentuk awal dari **perangkat lunak sumber terbuka** (*open-source software*).

Juga yang sangat penting adalah aksesibilitas dan keterbacaan kode. Memiliki kernel kecil yang indah yang ditulis dalam C mengundang orang lain untuk bermain dengan kernel, menambahkan fitur baru dan keren. Misalnya, grup giat di Berkeley, yang dipimpin oleh **Bill Joy**, membuat distribusi yang luar biasa (**Berkeley Systems Distribution**, atau **BSD**) yang memiliki memori virtual canggih, sistem file, dan subsistem jaringan. Joy kemudian mendirikan **Sun Microsystems**.

Sayangnya, penyebaran UNIX sedikit melambat karena perusahaan mencoba menegaskan kepemilikan dan mendapat untung darinya, hasil yang tidak menguntungkan (tetapi umum) dari pengacara yang terlibat. Banyak perusahaan memiliki varian mereka sendiri: **SunOS** dari Sun Microsystems, **AIX** dari IBM, **HPUX** (a.k.a. “H-Pucks”) dari HP, dan **IRIX** dari SGI. Pertengkarannya antara AT&T/Bell Labs dan pemain lain ini memberikan awan gelap di atas UNIX, dan banyak yang bertanya-tanya apakah itu akan bertahan, terutama karena Windows diperkenalkan dan mengambil alih banyak pasar PC...

## Era Multiprogramming

Di mana sistem operasi benar-benar lepas landas adalah di era komputasi di luar mainframe, yaitu **komputer mini**. Mesin klasik seperti keluarga PDP dari Digital Equipment membuat komputer jauh lebih terjangkau; jadi, alih-alih memiliki satu mainframe per organisasi besar, sekarang kumpulan orang yang lebih kecil dalam suatu organisasi kemungkinan bisa memiliki komputer mereka sendiri. Tidak mengherankan, salah satu dampak utama dari penurunan biaya ini adalah peningkatan aktivitas pengembang; lebih banyak orang pintar mendapatkan tangan mereka di komputer dan dengan demikian membuat sistem komputer melakukan hal-hal yang lebih menarik dan indah.

Secara khusus, **multiprogramming** menjadi hal biasa karena keinginan untuk memanfaatkan sumber daya mesin dengan lebih baik. Alih-alih hanya menjalankan satu pekerjaan pada satu

waktu, OS akan memuat sejumlah pekerjaan ke dalam memori dan beralih dengan cepat di antara mereka, sehingga meningkatkan pemanfaatan CPU. Peralihan ini sangat penting karena perangkat I/O lambat; membiarkan program menunggu di CPU sementara I/O-nya sedang dilayani adalah pemborosan waktu CPU. Sebaliknya, mengapa tidak beralih ke pekerjaan lain dan menjalankannya sebentar?

Keinginan untuk mendukung multiprogramming dan tumpang tindih dengan adanya I/O dan interupsi memaksa inovasi dalam pengembangan konseptual sistem operasi di sepanjang sejumlah arah. Masalah seperti **perlindungan memori** menjadi penting; kami tidak ingin satu program dapat mengakses memori program lain. Memahami cara menangani masalah **konkurensi** yang diperkenalkan oleh multiprogramming juga kritis; memastikan OS berperilaku dengan benar meskipun ada interupsi adalah tantangan besar. Kami akan mempelajari masalah ini dan topik terkait nanti di buku ini.

Salah satu kemajuan praktis utama pada masa itu adalah pengenalan sistem operasi UNIX, terutama berkat Ken Thompson (dan Dennis Ritchie) di Bell Labs (ya, perusahaan telepon). UNIX mengambil banyak ide bagus dari sistem operasi yang berbeda (terutama dari Multics [O72], dan beberapa dari sistem seperti TENEX [B+72] dan Sistem Berbagi Waktu Berkeley [S68]), tetapi membuatnya lebih sederhana dan lebih mudah digunakan. Segera tim ini mengirimkan kaset berisi kode sumber UNIX kepada orang-orang di seluruh dunia, banyak dari mereka kemudian terlibat dan menambahkan ke sistem itu sendiri; lihat INFO TAMBAHAN untuk detail lebih lanjut<sup>10</sup>.

## Era Modern

Di luar komputer mini datang jenis mesin baru, lebih murah, lebih cepat, dan untuk massa: **komputer pribadi**, atau **PC** seperti yang kita sebut hari ini. Dipimpin oleh mesin awal Apple (mis., Apple II) dan IBM PC, jenis mesin baru ini akan segera menjadi kekuatan dominan dalam komputasi, karena biaya rendahnya memungkinkan satu mesin per desktop alih-alih komputer mini bersama per kelompok kerja.

Sayangnya, untuk sistem operasi, PC pada awalnya mewakili lompatan besar ke belakang, karena sistem awal lupa (atau tidak pernah tahu) pelajaran yang dipelajari di era komputer mini. Misalnya, sistem operasi awal seperti **DOS** (**Disk Operating System**, dari **Microsoft**) tidak menganggap perlindungan memori itu penting; jadi, aplikasi berbahaya (atau mungkin hanya diprogram dengan buruk) dapat mencoret-coret seluruh memori. Generasi pertama **Mac OS** (v9 dan sebelumnya) mengambil pendekatan koperasi untuk penjadwalan pekerjaan; jadi, utas yang secara tidak sengaja terjebak dalam loop tak terbatas dapat mengambil alih seluruh sistem, memaksa reboot. Daftar menyakitkan fitur OS yang hilang dalam generasi sistem ini panjang, terlalu panjang untuk diskusi penuh di sini.

Untungnya, setelah beberapa tahun menderita, fitur lama sistem operasi komputer mini mulai menemukan jalan mereka ke desktop. Misalnya, Mac OS X/macOS memiliki UNIX pada intinya, termasuk semua fitur yang diharapkan dari sistem yang matang seperti itu. Windows juga telah mengadopsi banyak ide hebat dalam sejarah komputasi, dimulai khususnya dengan Windows NT, lompatan besar ke depan dalam teknologi OS Microsoft. Bahkan ponsel saat ini

---

<sup>10</sup>Kami akan menggunakan INFO TAMBAHAN dan kotak teks terkait lainnya untuk menarik perhatian ke berbagai item yang tidak cukup sesuai dengan alur utama teks. Kadang-kadang, kami bahkan akan menggunakannya hanya untuk membuat lelucon, karena mengapa tidak bersenang-senang sedikit di sepanjang jalan? Ya, banyak leluconnya buruk.

menjalankan sistem operasi (seperti Linux) yang jauh lebih mirip dengan apa yang dijalankan komputer mini pada 1970-an daripada apa yang dijalankan PC pada 1980-an (syukurlah); senang melihat bahwa ide-ide bagus yang dikembangkan pada masa kejayaan pengembangan OS telah menemukan jalan mereka ke dunia modern. Lebih baik lagi adalah bahwa ide-ide ini terus berkembang, menyediakan lebih banyak fitur dan membuat sistem modern menjadi lebih baik bagi pengguna dan aplikasi.

## INFO TAMBAHAN: DAN KEMUDIAN DATANGLAH LINUX

Untungnya bagi UNIX, seorang peretas muda Finlandia bernama **Linus Torvalds** memutuskan untuk menulis versi UNIX-nya sendiri yang meminjam banyak pada prinsip dan ide di balik sistem asli, tetapi tidak dari basis kode, sehingga menghindari masalah legalitas. Dia meminta bantuan dari banyak orang lain di seluruh dunia, memanfaatkan alat GNU canggih yang sudah ada [G85], dan segera **Linux** lahir (serta gerakan perangkat lunak sumber terbuka modern).

Seiring era internet muncul, sebagian besar perusahaan (seperti Google, Amazon, Facebook, dan lainnya) memilih untuk menjalankan Linux, karena gratis dan dapat dengan mudah dimodifikasi sesuai kebutuhan mereka; memang, sulit membayangkan kesuksesan perusahaan-perusahaan baru ini seandainya sistem seperti itu tidak ada. Karena ponsel pintar menjadi platform dominan yang dihadapi pengguna, Linux menemukan benteng di sana juga (melalui Android), karena banyak alasan yang sama. Dan Steve Jobs membawa lingkungan operasi **NeXTStep** berbasis UNIX bersamanya ke Apple, sehingga membuat UNIX populer di desktop (meskipun banyak pengguna teknologi Apple mungkin bahkan tidak menyadari fakta ini). Jadi UNIX terus hidup, lebih penting hari ini daripada sebelumnya. Dewa-dewa komputasi, jika Anda percaya pada mereka, harus berterima kasih atas hasil yang luar biasa ini.

## 1.7 Ringkasan

Jadi, kita punya pengantar ke OS. Sistem operasi saat ini membuat sistem relatif mudah digunakan, dan hampir semua sistem operasi yang Anda gunakan saat ini telah dipengaruhi oleh perkembangan yang akan kita bahas di seluruh buku ini.

Sayangnya, karena keterbatasan waktu, ada sejumlah bagian dari OS yang tidak akan kita bahas dalam buku ini. Misalnya, ada banyak kode **jaringan** dalam sistem operasi; kami menyerahkan kepada Anda untuk mengambil kelas jaringan untuk mempelajari lebih lanjut tentang itu. Demikian pula, perangkat **grafis** sangat penting; ambil kursus grafis untuk memperluas pengetahuan Anda ke arah itu. Akhirnya, beberapa buku sistem operasi berbicara banyak tentang **keamanan**; kami akan melakukannya dalam arti bahwa OS harus memberikan perlindungan antara program yang berjalan dan memberi pengguna kemampuan untuk melindungi file mereka, tetapi kami tidak akan menyelidiki masalah keamanan yang lebih dalam yang mungkin ditemukan dalam kursus keamanan.

Namun, ada banyak topik penting yang akan kami bahas, termasuk dasar-dasar virtualisasi CPU dan memori, konkurenси, dan persistensi melalui perangkat dan sistem file. Jangan khawatir! Meskipun ada banyak materi yang harus dibahas, sebagian besarnya cukup keren, dan di ujung jalan, Anda akan memiliki apresiasi baru tentang bagaimana sistem komputer

benar-benar bekerja. Sekarang mulailah bekerja!

## Referensi

- [BS+09] “Tolerating File-System Mistakes with EnvyFS” by L. Bairavasundaram, S. Sundararaman, A. Arpaci-Dusseau, R. Arpaci-Dusseau. USENIX '09, San Diego, CA, June 2009.
- [BH00] “The Evolution of Operating Systems” by P. Brinch Hansen. In 'Classic Operating Systems: From Batch Processing to Distributed Systems.' Springer-Verlag, New York, 2000.
- [B+72] “TENEX, A Paged Time Sharing System for the PDP-10” by D. Bobrow, J. Burchfiel, D. Murphy, R. Tomlinson. CACM, Volume 15, Number 3, March 1972.
- [B75] “The Mythical Man-Month” by F. Brooks. Addison-Wesley, 1975.
- [BOH10] “Computer Systems: A Programmer’s Perspective” by R. Bryant and D. O’Hallaron. Addison-Wesley, 2010.
- [G85] “The GNU Manifesto” by R. Stallman. 1985. [www.gnu.org/gnu/manifesto.html](http://www.gnu.org/gnu/manifesto.html).
- [K+61] “One-Level Storage System” by T. Kilburn, D.B.G. Edwards, M.J. Lanigan, F.H. Sumner. IRE Transactions on Electronic Computers, April 1962.
- [L78] “The Manchester Mark I and Atlas: A Historical Perspective” by S. H. Lavington. Communications of the ACM, Volume 21:1, January 1978.
- [O72] “The Multics System: An Examination of its Structure” by Elliott Organick. MIT Press, 1972.
- [PP03] “Introduction to Computing Systems: From Bits and Gates to C and Beyond” by Yale N. Patt, Sanjay J. Patel. McGraw-Hill, 2003.
- [RT74] “The UNIX Time-Sharing System” by Dennis M. Ritchie, Ken Thompson. CACM, Volume 17: 7, July 1974.
- [S68] “SDS 940 Time-Sharing System” by Scientific Data Systems. TECHNICAL MANUAL, SDS 90 11168, August 1968.
- [SS+10] “Membrane: Operating System Support for Restartable File Systems” by S. Sundararaman, S. Subramanian, A. Rajimwale, A. Arpaci-Dusseau, R. Arpaci-Dusseau, M. Swift. FAST '10, San Jose, CA, February 2010.

## Pekerjaan Rumah

Sebagian besar (dan akhirnya, semua) bab dari buku ini memiliki bagian pekerjaan rumah di bagian akhir. Mengerjakan pekerjaan rumah ini penting, karena masing-masing memungkinkan Anda, pembaca, mendapatkan lebih banyak pengalaman dengan konsep-konsep yang disajikan dalam bab ini.

Ada dua jenis pekerjaan rumah. Yang pertama didasarkan pada **simulasi**. Simulasi sistem komputer hanyalah program sederhana yang berpura-pura melakukan beberapa bagian menarik dari apa yang dilakukan sistem nyata, dan kemudian melaporkan beberapa metrik output untuk menunjukkan bagaimana sistem berperilaku. Misalnya, simulator hard drive mungkin mengambil serangkaian permintaan, mensimulasikan berapa lama waktu yang dibutuhkan untuk dilayani oleh hard drive dengan karakteristik kinerja tertentu, dan kemudian melaporkan latensi rata-rata permintaan.

Hal keren tentang simulasi adalah mereka membiarkan Anda dengan mudah mengeksplorasi bagaimana sistem berperilaku tanpa kesulitan menjalankan sistem nyata. Memang, mereka bahkan membiarkan Anda membuat sistem yang tidak mungkin ada di dunia nyata (misalnya, hard drive dengan kinerja yang sangat cepat tak terbayangkan), dan dengan demikian melihat potensi dampak teknologi masa depan.

Tentu saja, simulasi bukan tanpa kelemahan. Secara alami, simulasi hanyalah perkiraan tentang bagaimana sistem nyata berperilaku. Jika aspek penting dari perilaku dunia nyata dihilangkan, simulasi akan melaporkan hasil yang buruk. Jadi, hasil dari simulasi harus selalu diperlakukan dengan curiga. Pada akhirnya, bagaimana sistem berperilaku di dunia nyata adalah yang terpenting.

Jenis pekerjaan rumah kedua membutuhkan interaksi dengan **kode dunia nyata** (*real-world code*). Beberapa pekerjaan rumah ini berfokus pada pengukuran, sementara yang lain hanya memerlukan pengembangan dan eksperimen skala kecil. Keduanya hanyalah upaya kecil ke dunia yang lebih besar yang harus Anda masuki, yaitu cara menulis kode sistem dalam C pada sistem berbasis UNIX. Memang, proyek skala besar, yang melampaui pekerjaan rumah ini, diperlukan untuk mendorong Anda ke arah ini; jadi, di luar hanya mengerjakan pekerjaan rumah, kami sangat menyarankan Anda melakukan proyek untuk memantapkan keterampilan sistem Anda. Lihat halaman ini (<https://github.com/remzi-arpacidusseau/ostep-projects>) untuk beberapa proyek.

Untuk melakukan pekerjaan rumah ini, Anda kemungkinan harus berada di mesin berbasis UNIX, menjalankan Linux, macOS, atau sistem serupa. Itu juga harus memiliki kompiler C yang diinstal (mis., **gcc**) serta Python. Anda juga harus tahu cara mengedit kode di editor kode nyata semacam itu.

## Abstraksi: Proses

Dalam bab ini, kita akan membahas salah satu abstraksi paling mendasar yang disediakan oleh OS untuk pengguna: **proses**. Definisi dari proses, secara informal, cukup sederhana: proses adalah **program yang sedang berjalan** (*running program*). Program itu sendiri adalah hal yang mati: ia hanya diam di sana di disk, sekumpulan instruksi (dan mungkin beberapa data statis), menunggu untuk beraksi. Adalah sistem operasi yang mengambil byte-byte ini dan membuatnya berjalan, mengubah program menjadi sesuatu yang berguna.

Seringkali terjadi bahwa pengguna ingin menjalankan lebih dari satu program sekaligus; misalnya, Anda mungkin ingin menjalankan browser web, klien email, pemutar musik, dan sebagainya pada saat yang bersamaan. Bahkan dalam sistem yang hanya memiliki satu CPU fisik, OS menciptakan ilusi bahwa ada banyak CPU virtual, sehingga memungkinkan semua program ini berjalan seolah-olah mereka memiliki prosesor mereka sendiri. Teknik ini disebut **virtualisasi CPU** (atau *time sharing*), dan ini adalah fokus utama dari bagian pertama buku ini.

Untuk mengimplementasikan virtualisasi CPU, dan melakukannya dengan baik, OS memerlukan dukungan tingkat rendah dari perangkat keras (misalnya, mekanisme untuk beralih antar proses) serta beberapa kebijakan tingkat tinggi untuk membuat keputusan cerdas (misalnya, proses mana yang harus dijalankan selanjutnya?).

### INTI MASALAH: BAGAIMANA MENYEDIAKAN ILUSI BANYAK CPU?

Bagaimana OS menyediakan ilusi dari banyak CPU? Meskipun hanya ada sedikit CPU fisik yang tersedia, bagaimana OS dapat memberikan ilusi bahwa ada jumlah CPU virtual yang hampir tak terbatas?

Jawabannya, seperti yang akan kita lihat, adalah dengan memvirtualisasi CPU. Dengan menjalankan satu proses, lalu menghentikannya dan menjalankan yang lain, dan seterusnya, OS dapat mempromosikan ilusi bahwa ada banyak CPU virtual yang ada ketika pada kenyataannya hanya ada satu (atau beberapa) CPU fisik.

## 2.1 Abstraksi: Sebuah Proses

Untuk memahami apa yang membentuk sebuah proses, kita harus memahami **state mesin** (*machine state*): apa yang dapat dibaca atau diperbarui oleh program saat sedang berjalan. Pada waktu tertentu, bagian mana dari mesin yang penting bagi eksekusi program ini?

Salah satu komponen jelas dari state mesin adalah **memori**. Instruksi terletak di memori; data yang dibaca atau ditulis oleh program yang sedang berjalan juga ada di memori. Jadi **ruang alamat** (*address space*) yang dapat diakses oleh proses adalah bagian dari proses tersebut.

Bagian lain dari state mesin adalah **register**. Banyak instruksi secara eksplisit membaca atau memperbarui register; oleh karena itu, mereka jelas penting bagi eksekusi proses. Perhatikan bahwa ada beberapa register khusus yang membentuk bagian dari state mesin ini. Misalnya, **program counter** (PC) (kadang-kadang disebut *instruction pointer* atau IP) memberi tahu kita instruksi mana dari program yang akan dieksekusi selanjutnya; demikian pula **stack pointer** dan **frame pointer** terkait digunakan untuk mengelola tumpukan (*stack*) fungsi parameter, variabel lokal, dan alamat pengembalian.

Akhirnya, program sering mengakses informasi persisten yang disimpan dalam perangkat I/O juga. Daftar file yang saat ini dibuka oleh proses, misalnya, juga merupakan bagian dari proses.

## 2.2 API Proses

Hampir semua sistem operasi modern menyediakan antarmuka dasar yang sama untuk melakukan hal-hal terkait proses:

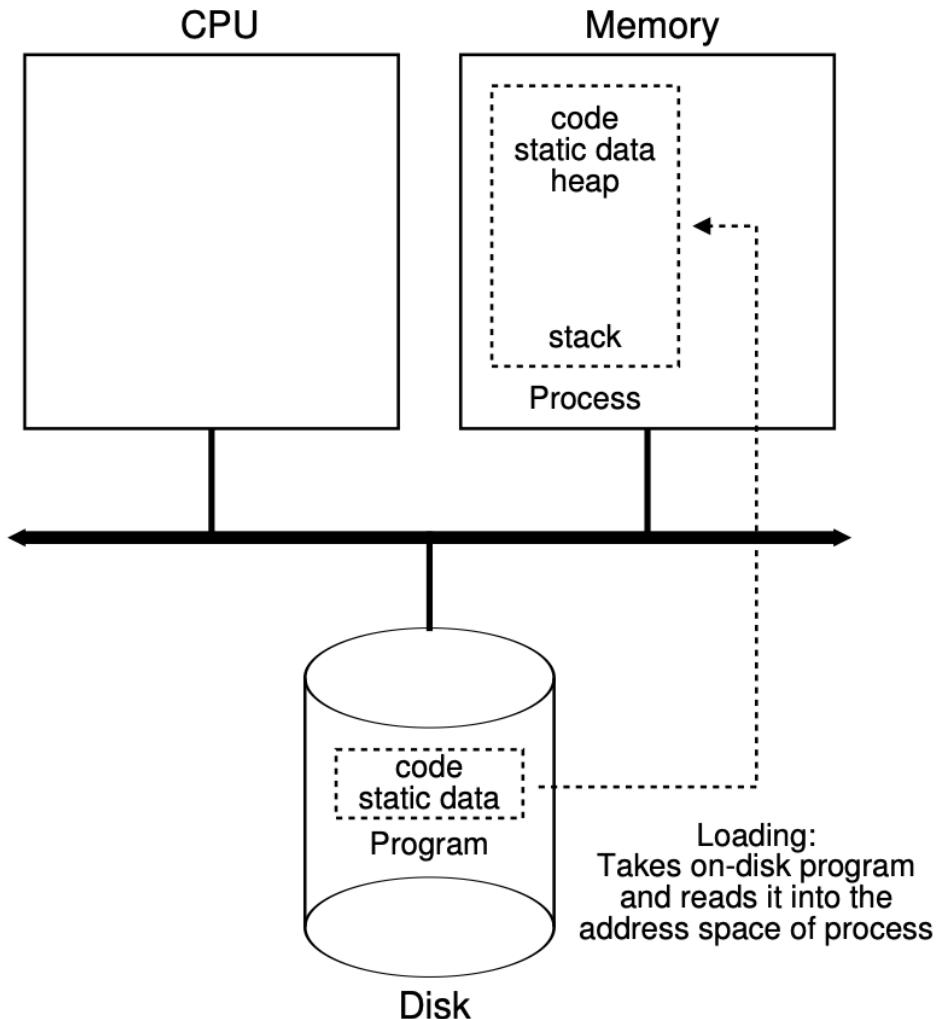
- **Create:** Harus ada metode untuk membuat proses baru. Ketika Anda mengetik perintah ke dalam shell, atau mengklik ganda ikon aplikasi, OS dipanggil untuk membuat proses baru untuk menjalankan program yang Anda indikasikan.
- **Destroy:** Karena ada antarmuka untuk pembuatan proses, harus ada juga antarmuka untuk menghancurnya secara paksa. Tentu saja, banyak proses akan berjalan dan keluar dengan sendirinya ketika selesai; namun, ketika mereka tidak (mis., bug, loop tak terbatas), pengguna mungkin ingin mematikannya.
- **Wait:** Kadang-kadang berguna untuk menunggu proses berhenti berjalan, jadi beberapa jenis antarmuka tunggu sering disediakan.
- **Miscellaneous Control:** Selain membunuh atau menunggu proses, ada kontrol lain yang mungkin berguna. Misalnya, sebagian besar sistem operasi menyediakan semacam metode untuk menangguhkan proses (menghentikannya berjalan untuk sementara waktu) dan kemudian melanjutkannya (melanjutkannya berjalan).
- **Status:** Biasanya ada antarmuka untuk mendapatkan beberapa informasi status tentang proses juga, seperti berapa lama proses telah berjalan, atau dalam keadaan apa proses itu sekarang.

## 2.3 Pembuatan Proses: Lebih Detail

Satu misteri kecil adalah bagaimana program diubah menjadi proses. Secara khusus, bagaimana OS mengambil data program (kode dan data statis) yang ada di disk dan membuatnya

berjalan?

Hal pertama yang harus dilakukan OS untuk menjalankan program adalah **memuat** (*load*) kode dan data statis apa pun (mis., variabel yang diinisialisasi) ke dalam memori, ke dalam ruang alamat proses. Program awalnya berada di disk (atau, dalam beberapa sistem modern, flash-based SSD) dalam beberapa jenis format yang dapat dieksekusi.



Gambar 2.1: Memuat: Dari Program Ke Proses

Setelah kode dan data statis dimuat ke dalam memori, ada beberapa hal lain yang perlu dilakukan OS sebelum menjalankan proses. Sejumlah memori harus dialokasikan untuk **stack** (tumpukan) run-time program. OS juga mungkin mengalokasikan memori untuk **heap** program. Heap digunakan untuk data yang diminta secara eksplisit secara dinamis (`malloc()` dalam C, atau `new` dalam C++ atau Java).

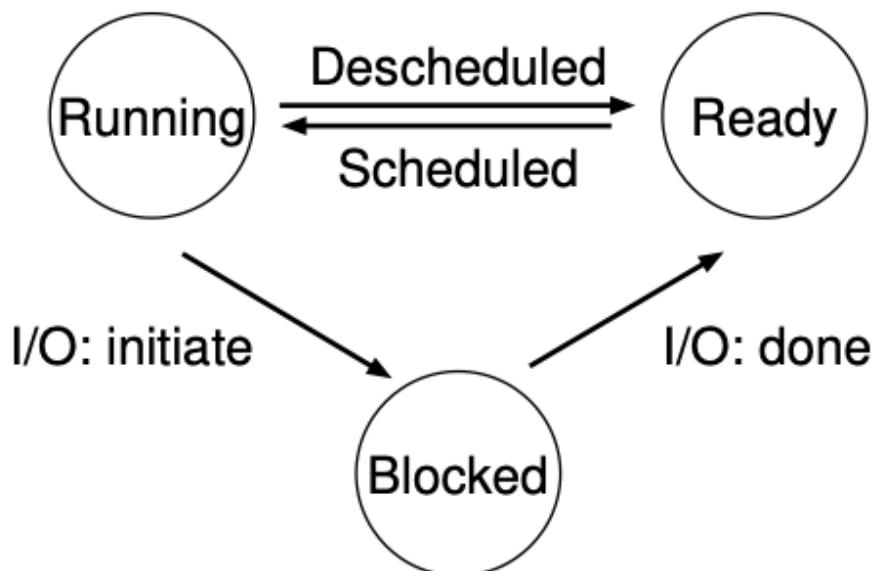
OS juga akan melakukan beberapa tugas inisialisasi lainnya, terutama terkait dengan input/output (I/O). Misalnya, dalam sistem UNIX, setiap proses secara default memiliki tiga deskriptor file terbuka, untuk input standar, output standar, dan kesalahan standar.

Akhirnya, OS menyelesaikan pengaturan program dan melompat ke rutinitas `main()`. Dengan melompat ke `main()`, OS mentransfer kendali CPU ke proses yang baru dibuat, dan dengan demikian program mulai dieksekusi.

## 2.4 State Proses

Sekarang kita memiliki gagasan tentang apa itu proses, dan bagaimana proses itu dibuat, mari kita bicara tentang berbagai **keadaan (states)** yang dapat dialami proses pada waktu tertentu. Gagasan awalnya sederhana. Dalam model yang disederhanakan, sebuah proses dapat berada dalam satu dari tiga keadaan:

- **Running (Berjalan):** Dalam keadaan berjalan, proses sedang berjalan di prosesor. Ini berarti proses sedang mengeksekusi instruksi.
- **Ready (Siap):** Dalam keadaan siap, proses siap dijalankan tetapi karena alasan tertentu OS telah memilih untuk tidak menjalankannya pada saat ini.
- **Blocked (Terblokir):** Dalam keadaan terblokir, proses telah melakukan beberapa jenis operasi yang membuatnya tidak siap untuk berjalan sampai beberapa peristiwa lain terjadi. Contoh umum: ketika proses memulai I/O ke disk, ia menjadi terblokir dan dengan demikian proses lain dapat menggunakan prosesor.



Gambar 2.2: Proses: Transisi State

Tabel 2.1: Pelacakan State Proses: CPU Saja (*Tracing Process State: CPU Only*)

Waktu	Proses 0	Proses 1	Catatan
1	Running	Ready	-
2	Running	Ready	-
3	Running	Ready	-
4	Running	Ready	Proses 0 selesai
5	-	Running	-
6	-	Running	-
7	-	Running	-
8	-	Running	Proses 1 selesai

Penjadwalan transisi antar keadaan ini adalah inti dari pekerjaan OS.

## 2.5 Struktur Data

OS adalah program, dan seperti program lainnya, OS memiliki beberapa struktur data kunci yang melacak berbagai bagian informasi yang relevan. Untuk melacak keadaan setiap proses, misalnya, OS kemungkinan akan menyimpan semacam **daftar proses** (*process list*) untuk semua proses yang siap dan beberapa informasi tambahan untuk melacak proses mana yang sedang berjalan. OS juga harus memiliki cara untuk melacak proses yang diblokir.

Mungkin struktur data yang paling penting adalah **blok kontrol proses** (*Process Control Block* atau **PCB**), yang berisi informasi tentang setiap proses. PCB akan menyimpan register proses (isi prosesor) ketika proses tidak berjalan, sehingga ketika OS melanjutkan proses tersebut, OS dapat memuat kembali register-register ini dan melanjutkan eksekusi seolah-olah proses tidak pernah dihentikan.

Struktur data ini memungkinkan OS untuk dengan anggun menangani banyak proses sekali-gus, menyimpan dan memulihkan keadaan mereka sesuai kebutuhan untuk menciptakan ilusi konkurensi.

## 2.6 Ringkasan

Dalam bab ini, kita telah memperkenalkan abstraksi paling dasar dari OS: proses. Ini cukup sederhana: itu hanya program yang sedang berjalan. Namun, untuk mengimplementasikan proses, OS harus melakukan banyak hal: memuat kode program ke dalam memori, menangani setup stack dan heap, dan melakukan tugas-tugas terkait lainnya. OS juga harus mengelola keadaan proses dan beralih di antara mereka untuk memvirtualisasi CPU. Di bab-bab selanjutnya, kita akan melihat lebih dalam mekanisme dasar yang memungkinkan hal ini terjadi.



## Antarmuka API Proses

Pada bab sebelumnya, kita membahas abstraksi proses dan bagaimana OS menciptakan ilusi banyak CPU. Pada bab ini, fokus kita adalah antarmuka pemrograman yang digunakan aplikasi untuk berinteraksi dengan proses di sistem UNIX: bagaimana proses dibuat, bagaimana proses menunggu proses lain, dan bagaimana program baru dijalankan dalam ruang proses yang sudah ada.

Secara praktis, ada tiga system call yang menjadi fondasi API proses:

- `fork()` untuk membuat proses baru,
- `wait()` / `waitpid()` untuk menunggu proses anak selesai,
- `exec()` (keluarga `exec`) untuk mengganti citra program yang sedang berjalan.

### INTI MASALAH: BAGAIMANA PROGRAM MEMBUAT DAN MENGELOLA PROSES?

Jika proses adalah abstraksi utama untuk virtualisasi CPU, maka bagaimana aplikasi benar-benar membuat proses baru? Bagaimana hubungan induk-anak dibentuk? Bagaimana sinkronisasi dilakukan agar proses induk tahu kapan anak selesai? Dan bagaimana proses memulai program yang berbeda tanpa membuat proses dari nol?

### 3.1 System Call `fork()`

`fork()` membuat salinan proses pemanggil. Setelah pemanggilan berhasil, akan ada dua proses yang melanjutkan eksekusi dari baris setelah `fork()`: proses induk (*parent*) dan proses anak (*child*).

Perilaku kunci dari `fork()` adalah nilai kembalinya:

- Pada proses *child*, nilai kembali `fork()` adalah 0.
- Pada proses *parent*, nilai kembali `fork()` adalah PID anak (bilangan positif).

- Jika gagal, fork() mengembalikan nilai negatif.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4
5 int main(void) {
6     printf("halo (pid:%d)\n", (int)getpid());
7
8     int rc = fork();
9     if (rc < 0) {
10         fprintf(stderr, "fork gagal\n");
11         exit(1);
12     } else if (rc == 0) {
13         printf("saya child (pid:%d)\n", (int)getpid());
14     } else {
15         printf("saya parent dari child %d (pid:%d)\n", rc, (int)
16             getpid());
17     }
18
19     return 0;
}

```

Listing 3.1: Contoh dasar penggunaan fork()

Karena parent dan child berjalan secara konkuren, urutan keluaran keduanya tidak dijamin sama di setiap eksekusi. Inilah contoh awal bahwa **penjadwal OS** menentukan siapa yang mendapat CPU lebih dulu.

## 3.2 Ruang Alamat Setelah fork()

Meskipun fork() terlihat seperti menyalin seluruh proses, implementasi modern biasanya menggunakan teknik *copy-on-write* agar efisien. Secara semantik, parent dan child memiliki ruang alamat terpisah: perubahan variabel di child tidak mengubah salinan variabel di parent, dan sebaliknya.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4
5 int x = 100;
6
7 int main(void) {
8     int rc = fork();
9     if (rc < 0) {
10         exit(1);
11     } else if (rc == 0) {
12         x = 200;
13         printf("child: x=%d (pid:%d)\n", x, (int)getpid());
14     } else {
15         x = 300;
16         printf("parent: x=%d (pid:%d)\n", x, (int)getpid());
}

```

```

17     }
18     return 0;
19 }
```

Listing 3.2: Variabel yang tampak sama tetapi hidup di proses berbeda

Kode di atas menegaskan bahwa proses adalah unit isolasi: masing-masing memiliki konteks eksekusi sendiri (register, ruang alamat, status file, dan metadata proses).

### 3.3 System Call wait() dan waitpid()

Sering kali parent perlu menunggu child selesai sebelum melanjutkan. Untuk itu, UNIX menyediakan `wait()` atau `waitpid()`.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/wait.h>
4 #include <unistd.h>
5
6 int main(void) {
7     int rc = fork();
8     if (rc < 0) {
9         exit(1);
10    } else if (rc == 0) {
11        printf("child berjalan (pid:%d)\n", (int)getpid());
12    } else {
13        int wc = wait(NULL);
14        printf("parent (pid:%d): child %d selesai\n", (int)getpid(),
15               , wc);
16    }
17    return 0;
}
```

Listing 3.3: Parent menunggu child selesai

Dengan `wait()`, parent diblokir sampai salah satu child selesai. Ini adalah pola sinkronisasi sederhana tetapi sangat umum, misalnya pada shell ketika menjalankan program di mode *foreground*.

### 3.4 System Call Keluarga exec()

`fork()` membuat proses baru yang awalnya mengeksekusi kode yang sama dengan parent. Namun, pada banyak kasus kita ingin child menjalankan program **lain**. Di sinilah `exec()` digunakan.

Keluarga `exec` (misalnya `exec1()`, `execvp()`) **mengganti** citra proses saat ini dengan program baru. Jika berhasil, kode lama tidak dilanjutkan.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
```

```

4 #include <sys/wait.h>
5
6 int main(void) {
7     int rc = fork();
8     if (rc < 0) {
9         exit(1);
10    } else if (rc == 0) {
11        char *myargs[3];
12        myargs[0] = "wc";
13        myargs[1] = "chapter3-book-translate.tex";
14        myargs[2] = NULL;
15        execvp(myargs[0], myargs);
16        printf("baris ini tidak tercetak jika exec berhasil\n");
17    } else {
18        wait(NULL);
19        printf("parent: child selesai\n");
20    }
21    return 0;
22 }
```

Listing 3.4: Menggabungkan fork() dan execvp()

Pola fork() lalu exec() adalah dasar implementasi shell UNIX. Shell membuat child, child memanggil exec(), lalu parent menunggu (atau tidak menunggu untuk *background job*).

## 3.5 Mengapa API Proses Dirancang Seperti Ini?

Desain ini terkenal karena sederhana namun sangat fleksibel:

- fork() memisahkan langkah *membuat proses* dari langkah *menjalankan program baru*.
- Antara fork() dan exec(), child bisa melakukan pengaturan lanjutan: mengubah direktori kerja, mengatur *file descriptor*, mengalihkan I/O, dan sebagainya.
- wait() memberi kontrol sinkronisasi yang jelas kepada parent.

Kombinasi ini membuat banyak fitur sistem mudah dibangun, seperti pipa antar-proses, redireksi file, *job control*, dan daemon.

## 3.6 Redireksi Output dan Pipe

Pemisahan fork() dan exec() memungkinkan shell menyiapkan lingkungan child sebelum program baru dijalankan. Contoh paling umum adalah redireksi keluaran:

- Shell membuat child dengan fork().
- Di child, STDOUT\_FILENO ditutup.
- Shell membuka file tujuan, sehingga descriptor terendah yang kosong (biasanya 1) dipakai untuk file tersebut.
- Child memanggil exec(), dan keluaran program otomatis mengarah ke file.

```

1 #include <fcntl.h>
2 #include <stdlib.h>
3 #include <sys/wait.h>
4 #include <unistd.h>
5
6 int main(void) {
7     int rc = fork();
8     if (rc < 0) return 1;
9
10    if (rc == 0) {
11        close(STDOUT_FILENO);
12        open("./hasil.txt", O_CREAT | O_WRONLY | O_TRUNC, 0700);
13
14        char *args[] = {"wc", "chapter3-book-translate.tex", NULL};
15        execvp(args[0], args);
16        exit(1);
17    }
18
19    wait(NULL);
20    return 0;
21 }
```

Listing 3.5: Sketsa redireksi sebelum exec()

Konsep yang sama dipakai untuk *pipe*: keluaran proses pertama dihubungkan ke masukan proses kedua melalui pipe() dan pengaturan descriptor file. Karena itu shell bisa membangun rantai perintah seperti grep -o kata file | wc -l.

## 3.7 Kontrol Proses Dengan Sinyal

Selain membuat dan menunggu proses, UNIX menyediakan mekanisme kontrol lewat *signal*. Dengan kill() (atau utilitas shell seperti kill/killall), proses dapat menerima sinyal untuk:

- menghentikan proses (misalnya SIGINT),
- menjeda proses (SIGTSTP),
- melanjutkan proses yang dijeda (SIGCONT),
- atau tindakan lain tergantung jenis sinyal.

Pada shell interaktif, kombinasi tombol seperti Ctrl-C dan Ctrl-Z pada dasarnya mengirim sinyal ke proses *foreground*. Program juga bisa memasang handler (misalnya lewat signal()) untuk merespons sinyal tertentu.

## 3.8 Pengguna, Hak Akses, dan Superuser

Kontrol proses selalu terkait model pengguna (*user*) di sistem operasi:

- User biasa umumnya hanya boleh mengontrol proses miliknya sendiri.

- OS memakai batasan ini untuk keamanan dan isolasi antar-user.
- Superuser (root) dapat mengontrol proses user lain dan menjalankan operasi administratif.

Kewenangan root kuat, tetapi berisiko jika salah pakai. Praktik yang aman adalah memakai akun biasa untuk pekerjaan harian dan naik hak akses hanya saat benar-benar perlu.

## 3.9 Alat Praktis untuk Observasi Proses

Beberapa tool command-line yang sering dipakai:

- ps: melihat daftar proses yang sedang berjalan.
- top: memantau pemakaian CPU/memori secara real-time.
- kill / killall: mengirim sinyal ke proses.

Tool-tool ini melengkapi pemahaman API proses: bukan hanya bagaimana proses dibuat, tetapi juga bagaimana proses dipantau dan dikendalikan saat sistem berjalan.

### **INTI MASALAH: BAGAIMANA SHELL MENJALANKAN PERINTAH?**

Shell modern bergantung pada tiga primitive ini: `fork()`, `exec()`, dan `wait()`. Dengan primitive sederhana tersebut, shell dapat mengeksekusi perintah pengguna, mengatur proses latar depan/latar belakang, serta menghubungkan keluaran dan masukan antar-program.

## 3.10 Ringkasan

Bab ini memperkenalkan API proses pada sistem UNIX. Kita mempelajari bahwa `fork()` membuat proses anak, `wait()` menyinkronkan parent dengan child, dan `exec()` mengganti program yang dijalankan oleh suatu proses. Kita juga melihat mengapa pemisahan `fork()` dan `exec()` penting untuk membangun fitur shell seperti redireksi dan *pipe*, bagaimana proses dikontrol melalui sinyal, serta bagaimana model user/root membatasi hak kontrol proses. Secara praktis, kombinasi API dan tool observasi seperti `ps`, `top`, dan `kill` adalah fondasi kerja sehari-hari dalam sistem operasi UNIX modern.

# Penjadwalan: Pengenalan

Sekarang, mekanisme tingkat rendah dalam menjalankan proses (misalnya, *context switching*) seharusnya sudah jelas; jika belum, kembali satu atau dua bab, dan baca kembali penjelasan tentang bagaimana hal tersebut bekerja. Namun, kita belum memahami **kebijakan** tingkat tinggi yang digunakan oleh penjadwal OS. Kita sekarang akan membahas hal itu, menyajikan serangkaian **kebijakan penjadwalan** (kadang-kadang disebut *disiplin*) yang telah dikembangkan oleh berbagai orang cerdas dan pekerja keras selama bertahun-tahun.

Asal-usul penjadwalan, pada kenyataannya, mendahului sistem komputer; pendekatan awal diambil dari bidang **manajemen operasi** dan diterapkan pada komputer. Kenyataan ini seharusnya tidak mengejutkan: lini perakitan dan banyak usaha manusia lainnya juga membutuhkan penjadwalan, dan banyak kekhawatiran yang sama ada di dalamnya, termasuk keinginan tajam untuk efisiensi.

## INTI MASALAH: BAGAIMANA MENGEMBANGKAN KEBIJAKAN PENJADWALAN

Bagaimana kita harus mengembangkan kerangka dasar untuk berpikir tentang kebijakan penjadwalan? Apa saja asumsi kunci? Metrik apa yang penting? Pendekatan dasar apa yang telah digunakan dalam sistem komputer paling awal?

### 4.1 Asumsi Beban Kerja

Sebelum masuk ke berbagai kemungkinan kebijakan, mari kita terlebih dahulu membuat sejumlah asumsi yang menyederhanakan tentang proses-proses yang berjalan dalam sistem, yang kadang-kadang secara kolektif disebut **beban kerja** (*workload*). Menentukan beban kerja adalah bagian penting dari membangun kebijakan, dan semakin banyak yang Anda ketahui tentang beban kerja, semakin baik kebijakan Anda dapat disesuaikan.

Asumsi beban kerja yang kita buat di sini sebagian besar tidak realistik, tetapi tidak apa-apa (untuk saat ini), karena kita akan melonggarkannya seiring berjalannya waktu, dan akhirnya mengembangkan apa yang akan kita sebut sebagai . . . (jeda dramatis) . . . sebuah disiplin

penjadwalan yang sepenuhnya operasional.

Kita akan membuat asumsi berikut tentang proses-proses, yang kadang-kadang disebut **pekerjaan** (*jobs*), yang berjalan dalam sistem:

1. Setiap pekerjaan berjalan untuk jumlah waktu yang sama.
2. Semua pekerjaan tiba pada waktu yang sama.
3. Setelah dimulai, setiap pekerjaan berjalan sampai selesai.
4. Semua pekerjaan hanya menggunakan CPU (yaitu, tidak melakukan I/O).
5. Waktu eksekusi setiap pekerjaan diketahui.

Kita mengatakan banyak dari asumsi ini tidak realistik, tetapi seperti halnya beberapa hewan “lebih setara dari yang lain” dalam *Animal Farm* karya Orwell [O45], beberapa asumsi lebih tidak realistik daripada yang lain dalam bab ini. Secara khusus, mungkin mengganggu Anda bahwa waktu eksekusi setiap pekerjaan diketahui: ini akan membuat penjadwal menjadi **maha tahu** (*omniscient*), yang meskipun akan hebat (mungkin), tidak mungkin terjadi dalam waktu dekat.

## 4.2 Metrik Penjadwalan

Selain membuat asumsi beban kerja, kita juga membutuhkan satu hal lagi untuk memungkinkan kita membandingkan kebijakan penjadwalan yang berbeda: sebuah **metrik penjadwalan**. Metrik hanyalah sesuatu yang kita gunakan untuk mengukur sesuatu, dan ada sejumlah metrik berbeda yang masuk akal dalam penjadwalan.

Untuk saat ini, mari kita juga menyederhanakan hidup kita dengan hanya memiliki satu metrik: **turnaround time** (waktu penyelesaian). Turnaround time dari sebuah pekerjaan didefinisikan sebagai waktu di mana pekerjaan selesai dikurangi waktu di mana pekerjaan tiba dalam sistem. Secara lebih formal:

$$T_{\text{turnaround}} = T_{\text{selesai}} - T_{\text{tiba}}$$

Karena kita telah mengasumsikan bahwa semua pekerjaan tiba pada waktu yang sama, untuk saat ini  $T_{\text{tiba}} = 0$  dan karenanya  $T_{\text{turnaround}} = T_{\text{selesai}}$ . Fakta ini akan berubah seiring kita melonggarkan asumsi-asumsi yang disebutkan di atas.

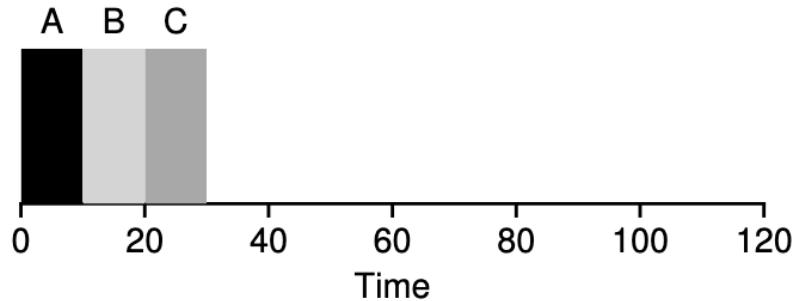
Anda perlu memperhatikan bahwa turnaround time adalah metrik **kinerja**, yang akan menjadi fokus utama kita di bab ini. Metrik lain yang menarik adalah **keadilan** (*fairness*), sebagaimana diukur (misalnya) oleh Indeks Keadilan Jain [J91]. Kinerja dan keadilan sering bertentangan dalam penjadwalan; sebuah penjadwal, misalnya, mungkin mengoptimalkan kinerja tetapi dengan biaya mencegah beberapa pekerjaan berjalan, sehingga menurunkan keadilan. Dilema ini menunjukkan kepada kita bahwa hidup tidak selalu sempurna.

## 4.3 First In, First Out (FIFO)

Algoritma paling dasar yang dapat kita implementasikan dikenal sebagai penjadwalan **First In, First Out (FIFO)** atau kadang-kadang **First Come, First Served (FCFS)**.

FIFO memiliki sejumlah properti positif: jelas sederhana dan dengan demikian mudah diimplementasikan. Dan, dengan asumsi kita, ia bekerja cukup baik.

Mari kita lihat contoh cepat bersama. Bayangkan tiga pekerjaan tiba dalam sistem, A, B, dan C, pada waktu yang hampir bersamaan ( $T_{tiba} \approx 0$ ). Karena FIFO harus menempatkan satu pekerjaan lebih dulu, mari kita asumsikan bahwa meskipun mereka semua tiba secara bersamaan, A tiba sedikit lebih dulu dari B yang tiba sedikit lebih dulu dari C. Asumsikan juga bahwa setiap pekerjaan berjalan selama 10 detik. Berapa rata-rata turnaround time untuk pekerjaan-pekerjaan ini?

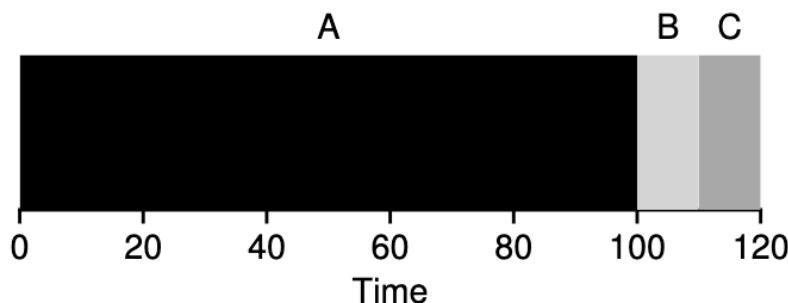


Gambar 4.1: Contoh Sederhana FIFO

Dari skenario ini, Anda dapat melihat bahwa A selesai pada 10, B pada 20, dan C pada 30. Jadi, rata-rata turnaround time untuk ketiga pekerjaan ini hanya  $\frac{10+20+30}{3} = 20$  detik. Menghitung turnaround time semudah itu.

Sekarang mari kita longgarkan salah satu asumsi kita. Secara khusus, mari kita longgarkan asumsi 1, dan dengan demikian tidak lagi mengasumsikan bahwa setiap pekerjaan berjalan untuk jumlah waktu yang sama. Bagaimana kinerja FIFO sekarang? Jenis beban kerja apa yang bisa Anda buat untuk membuat FIFO berkinerja buruk?

Mungkin Anda sudah mengetahuinya, tetapi untuk berjaga-jaga, mari kita lihat contoh untuk menunjukkan bagaimana pekerjaan dengan panjang berbeda dapat menyebabkan masalah untuk penjadwalan FIFO. Secara khusus, mari kita asumsikan lagi tiga pekerjaan (A, B, dan C), tetapi kali ini A berjalan selama 100 detik sementara B dan C berjalan masing-masing selama 10 detik.



Gambar 4.2: Mengapa FIFO Tidak Begitu Bagus

Pekerjaan A berjalan lebih dulu selama 100 detik penuh sebelum B atau C bahkan mendapat kesempatan untuk berjalan. Jadi, rata-rata turnaround time untuk sistem adalah tinggi: menyakitkan 110 detik ( $\frac{100+110+120}{3} = 110$ ).

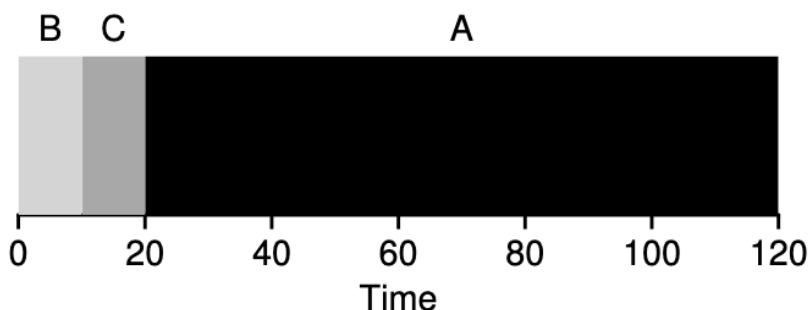
Masalah ini umumnya disebut sebagai efek **konvoi** (*convoy effect*) [B+79], di mana sejumlah konsumen sumber daya potensial yang relatif pendek mengantri di belakang konsumen sumber daya yang berat. Skenario penjadwalan ini mungkin mengingatkan Anda pada antrian tunggal di toko kelontong dan apa yang Anda rasakan ketika Anda melihat orang di depan Anda dengan tiga troli penuh persediaan dan buku cek keluar; ini akan memakan waktu lama.

### TIP: PRINSIP SJF

Shortest Job First mewakili prinsip penjadwalan umum yang dapat diterapkan pada sistem apa pun di mana turnaround time yang dirasakan per pelanggan (atau, dalam kasus kita, sebuah pekerjaan) penting. Pikirkan tentang antrian mana pun yang pernah Anda tunggu: jika perusahaan tersebut peduli dengan kepuasan pelanggan, kemungkinan mereka telah memperhitungkan SJF. Misalnya, toko kelontong biasanya memiliki jalur “sepuluh-item-atau-kurang” untuk memastikan bahwa pembeli dengan hanya beberapa barang untuk dibeli tidak terjebak di belakang keluarga yang mempersiapkan diri untuk bencana nuklir.

## 4.4 Shortest Job First (SJF)

Ternyata pendekatan yang sangat sederhana menyelesaikan masalah ini; sebenarnya ini adalah ide yang dicuri dari riset operasi [C54, PV56] dan diterapkan pada penjadwalan pekerjaan dalam sistem komputer. Disiplin penjadwalan baru ini dikenal sebagai **Shortest Job First (SJF)**, dan namanya seharusnya mudah diingat karena menjelaskan kebijakan dengan cukup lengkap: ia menjalankan pekerjaan terpendek lebih dulu, kemudian yang terpendek berikutnya, dan seterusnya.



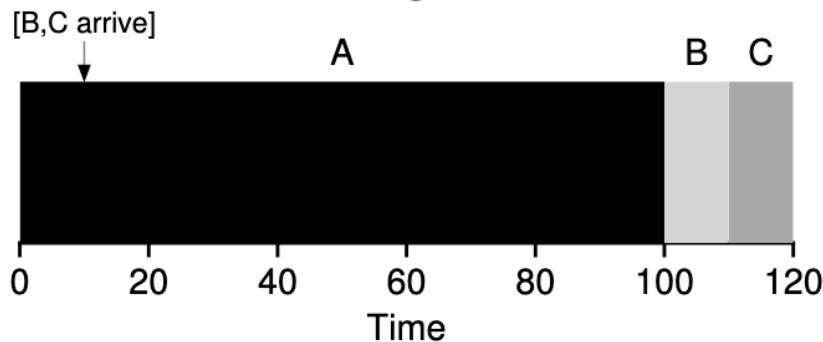
Gambar 4.3: Contoh Sederhana SJF

Mari kita ambil contoh di atas tetapi dengan SJF sebagai kebijakan penjadwalan kita. Dalam skenario ini, B dan C dijalankan sebelum A. Semoga jelas mengapa SJF berkinerja jauh lebih baik dalam hal rata-rata turnaround time. Hanya dengan menjalankan B dan C sebelum A, SJF mengurangi rata-rata turnaround dari 110 detik menjadi 50 detik ( $\frac{10+20+120}{3} = 50$ ), peningkatan lebih dari faktor dua.

Faktanya, dengan asumsi kita tentang pekerjaan yang semuanya tiba pada waktu yang sama, kita bisa membuktikan bahwa SJF memang merupakan algoritma penjadwalan yang **optimal**. Namun, Anda berada di kelas sistem, bukan teori atau riset operasi; tidak ada bukti yang diperbolehkan.

Jadi kita sampai pada pendekatan yang baik untuk penjadwalan dengan SJF, tetapi asumsi kita masih cukup tidak realistik. Mari kita longgarkan asumsi lain. Secara khusus, kita dapat menargetkan asumsi 2, dan sekarang mengasumsikan bahwa pekerjaan dapat tiba kapan saja alih-alih sekaligus. Masalah apa yang muncul dari ini?

Kita dapat mengilustrasikan masalah lagi dengan sebuah contoh. Kali ini, asumsikan A tiba pada  $t = 0$  dan perlu berjalan selama 100 detik, sedangkan B dan C tiba pada  $t = 10$  dan masing-masing perlu berjalan selama 10 detik. Dengan SJF murni, A mulai berjalan pada waktu 0 dan berjalan hingga selesai pada 100, lalu B berjalan 100–110, dan C berjalan 110–120.



Gambar 4.4: SJF Dengan Kedatangan Terlambat dari B dan C

### INFO TAMBAHAN: PENJADWAL PREEMPTIVE

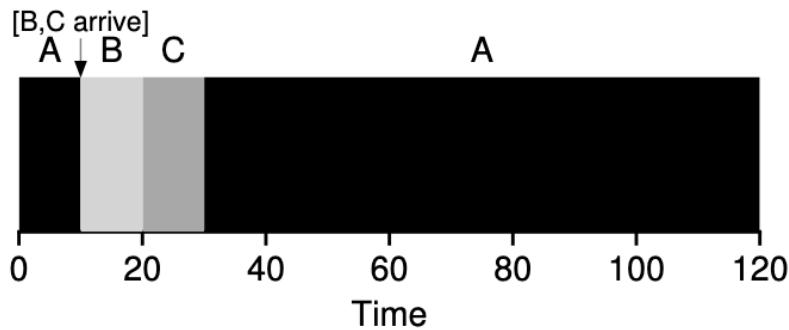
Pada masa lalu komputasi batch, sejumlah penjadwal *non-preemptive* dikembangkan; sistem semacam itu akan menjalankan setiap pekerjaan sampai selesai sebelum mempertimbangkan apakah akan menjalankan pekerjaan baru. Hampir semua penjadwal modern bersifat **preemptive**, dan cukup bersedia untuk menghentikan satu proses dari berjalan untuk menjalankan proses lain. Ini menyiratkan bahwa penjadwal menggunakan mekanisme yang kita pelajari sebelumnya; secara khusus, penjadwal dapat melakukan *context switch*, menghentikan sementara satu proses yang berjalan dan melanjutkan (atau memulai) proses lain.

Seperti yang Anda lihat, meskipun B dan C tiba tak lama setelah A, mereka masih dipaksa menunggu sampai A selesai, dan dengan demikian mengalami masalah konvoi yang sama. Rata-rata turnaround time untuk ketiga pekerjaan ini adalah 103,33 detik ( $\frac{100+(110-10)+(120-10)}{3} = 103,33$ ). Apa yang bisa dilakukan penjadwal?

## 4.5 Shortest Time-to-Completion First (STCF)

Untuk mengatasi masalah ini, kita perlu melonggarkan asumsi 3 (bahwa pekerjaan harus berjalan sampai selesai). Kita juga membutuhkan beberapa mekanisme dalam penjadwal itu sendiri. Seperti yang mungkin Anda duga, mengingat diskusi kita sebelumnya tentang *timer interrupt* dan *context switching*, penjadwal tentu bisa melakukan sesuatu yang lain ketika B dan C tiba: ia dapat **mendahului** (*preempt*) pekerjaan A dan memutuskan untuk menjalankan pekerjaan lain, mungkin melanjutkan A nanti. SJF menurut definisi kita adalah penjadwal *non-preemptive*, dan dengan demikian mengalami masalah yang dijelaskan di atas.

Untungnya, ada penjadwal yang melakukan persis itu: menambahkan *preemption* ke SJF, yang dikenal sebagai **Shortest Time-to-Completion First (STCF)** atau penjadwal **Preemptive Shortest Job First (PSJF)** [CK68]. Setiap kali pekerjaan baru masuk ke sistem, penjadwal STCF menentukan pekerjaan mana yang tersisa (termasuk pekerjaan baru) yang memiliki waktu tersisa paling sedikit, dan menjadwalkan pekerjaan tersebut. Jadi, dalam contoh kita, STCF akan mendahului A dan menjalankan B dan C sampai selesai; hanya ketika mereka selesai, waktu tersisa A akan dijadwalkan.



Gambar 4.5: Contoh Sederhana STCF

Hasilnya adalah rata-rata turnaround time yang jauh lebih baik: 50 detik ( $\frac{(120-0)+(20-10)+(30-10)}{3} = 50$ ). Dan seperti sebelumnya, dengan asumsi baru kita, STCF terbukti optimal.

## 4.6 Metrik Baru: Response Time

Jadi, jika kita mengetahui panjang pekerjaan, dan bahwa pekerjaan hanya menggunakan CPU, dan satu-satunya metrik kita adalah turnaround time, STCF akan menjadi kebijakan yang hebat. Faktanya, untuk sejumlah sistem komputasi batch awal, jenis algoritma penjadwalan ini masuk akal. Namun, pengenalan mesin *time-shared* mengubah segalanya. Sekarang pengguna akan duduk di terminal dan menuntut kinerja interaktif dari sistem juga. Dan dengan demikian, sebuah metrik baru lahir: **response time** (waktu respons).

Kita mendefinisikan response time sebagai waktu dari ketika pekerjaan tiba dalam sistem sampai pertama kali dijadwalkan. Secara lebih formal:

$$T_{\text{response}} = T_{\text{pertama\_dihadwalkan}} - T_{\text{tiba}}$$

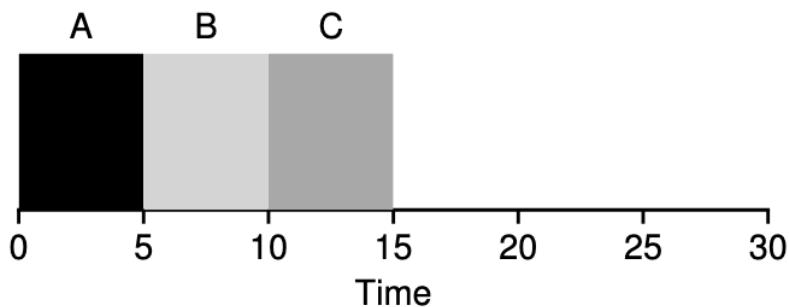
Seperti yang mungkin Anda pikirkan, STCF dan disiplin terkait tidak terlalu bagus untuk response time. Jika tiga pekerjaan tiba pada waktu yang sama, misalnya, pekerjaan ketiga harus menunggu dua pekerjaan sebelumnya berjalan secara keseluruhan sebelum dijadwalkan untuk pertama kalinya. Meskipun bagus untuk turnaround time, pendekatan ini cukup buruk untuk response time dan interaktivitas. Memang, bayangkan duduk di terminal, mengetik, dan harus menunggu 10 detik untuk melihat respons dari sistem hanya karena beberapa pekerjaan lain dijadwalkan di depan Anda: tidak terlalu menyenangkan.

Jadi, kita dihadapkan dengan masalah lain: bagaimana kita dapat membangun penjadwal yang sensitif terhadap response time?

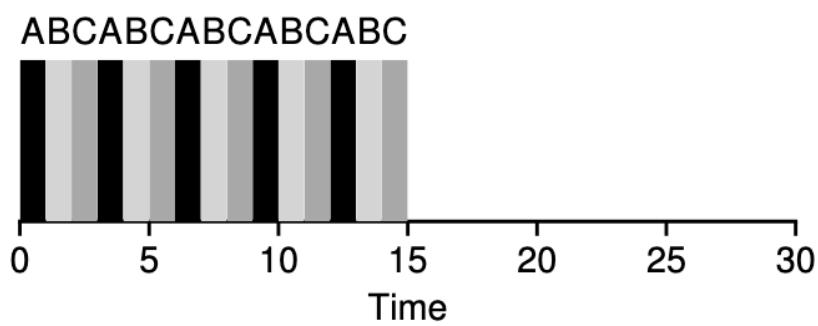
## 4.7 Round Robin

Untuk memecahkan masalah ini, kita akan memperkenalkan algoritma penjadwalan baru, yang secara klasik disebut sebagai penjadwalan **Round-Robin (RR)** [K64]. Ideanya sederhana: alih-alih menjalankan pekerjaan sampai selesai, RR menjalankan pekerjaan selama **irisan waktu** (*time slice*, kadang-kadang disebut **kuantum penjadwalan**) dan kemudian beralih ke pekerjaan berikutnya dalam antrian eksekusi. Ia berulang kali melakukan hal ini sampai pekerjaan-pekerjaan selesai. Untuk alasan ini, RR kadang-kadang disebut **time-slicing**. Perhatikan bahwa panjang irisan waktu harus merupakan kelipatan dari periode timer-interrupt; jadi jika timer menginterupsi setiap 10 milidetik, irisan waktu bisa 10, 20, atau kelipatan lain dari 10 ms.

Untuk memahami RR secara lebih rinci, mari kita lihat sebuah contoh. Asumsikan tiga pekerjaan A, B, dan C tiba pada waktu yang sama dalam sistem, dan masing-masing ingin berjalan selama 5 detik. Penjadwal SJF menjalankan setiap pekerjaan sampai selesai sebelum menjalankan yang lain (Gambar 4.6). Sebaliknya, RR dengan irisan waktu 1 detik akan berputar melalui pekerjaan dengan cepat: A, B, C, A, B, C, dst (Gambar 4.7).



Gambar 4.6: SJF Lagi (Buruk Untuk Response Time)



Gambar 4.7: Round Robin (Baik Untuk Response Time)

Rata-rata response time RR adalah:  $\frac{0+1+2}{3} = 1$  detik; untuk SJF, rata-rata response time adalah:  $\frac{0+5+10}{3} = 5$  detik.

Seperti yang Anda lihat, panjang irisan waktu sangat penting untuk RR. Semakin pendek, semakin baik kinerja RR di bawah metrik response-time. Namun, membuat irisan waktu terlalu pendek bermasalah: tiba-tiba biaya **context switching** akan mendominasi kinerja keseluruhan. Jadi, menentukan panjang irisan waktu menyajikan *trade-off* bagi desainer sis-

tem, membuatnya cukup panjang untuk **mengamortisasi** biaya peralihan tanpa membuatnya terlalu panjang sehingga sistem tidak lagi responsif.

## TIP: AMORTISASI DAPAT MENGURANGI BIAYA

Teknik umum amortisasi biasanya digunakan dalam sistem ketika ada biaya tetap untuk beberapa operasi. Dengan menanggung biaya itu lebih jarang (yaitu, dengan melakukan operasi lebih sedikit kali), total biaya untuk sistem berkurang. Misalnya, jika irisan waktu diatur ke 10 ms, dan biaya context-switch adalah 1 ms, kira-kira 10% waktu dihabiskan untuk context switching dan dengan demikian terbuang. Jika kita ingin mengamortisasi biaya ini, kita dapat meningkatkan irisan waktu, misalnya, menjadi 100 ms. Dalam kasus ini, kurang dari 1% waktu yang dihabiskan untuk context switching, dan dengan demikian biaya time-slicing telah diamortisasi.

Perhatikan bahwa biaya context switching tidak semata-mata muncul dari tindakan OS menyimpan dan mengembalikan beberapa register. Ketika program berjalan, mereka membangun banyak state di cache CPU, TLB, branch predictor, dan perangkat keras on-chip lainnya. Beralih ke pekerjaan lain menyebabkan state ini di-flush dan state baru yang relevan dengan pekerjaan yang sedang berjalan dibawa masuk, yang mungkin memiliki biaya kinerja yang nyata [MB91].

RR, dengan irisan waktu yang wajar, dengan demikian adalah penjadwal yang sangat baik jika response time adalah satu-satunya metrik kita. Tetapi bagaimana dengan turnaround time? Mari kita lihat contoh kita di atas lagi. A, B, dan C, masing-masing dengan waktu eksekusi 5 detik, tiba pada waktu yang sama, dan RR adalah penjadwal dengan irisan waktu 1 detik. Kita dapat melihat bahwa A selesai pada 13, B pada 14, dan C pada 15, untuk rata-rata 14. Cukup buruk!

Tidak mengherankan, RR memang adalah salah satu kebijakan terburuk jika turnaround time adalah metrik kita. Secara intuitif, ini masuk akal: apa yang dilakukan RR adalah meregangkan setiap pekerjaan selama mungkin, dengan hanya menjalankan setiap pekerjaan untuk sebentar sebelum beralih ke yang berikutnya. Karena turnaround time hanya peduli tentang kapan pekerjaan selesai, RR hampir **pesimal**, bahkan lebih buruk daripada FIFO sederhana dalam banyak kasus.

Secara lebih umum, kebijakan apa pun (seperti RR) yang **adil**, yaitu yang membagi CPU secara merata di antara proses-proses aktif pada skala waktu kecil, akan berkinerja buruk pada metrik seperti turnaround time. Memang, ini adalah *trade-off* yang melekat: jika Anda bersedia untuk tidak adil, Anda dapat menjalankan pekerjaan yang lebih pendek sampai selesai, tetapi biayanya adalah response time; jika sebaliknya Anda menghargai keadilan, response time akan berkurang, tetapi biayanya adalah turnaround time.

Kita telah mengembangkan dua jenis penjadwal. Jenis pertama (SJF, STCF) mengoptimalkan turnaround time, tetapi buruk untuk response time. Jenis kedua (RR) mengoptimalkan response time tetapi buruk untuk turnaround time. Dan kita masih memiliki dua asumsi yang perlu dilonggarkan: asumsi 4 (bahwa pekerjaan tidak melakukan I/O), dan asumsi 5 (bahwa waktu eksekusi setiap pekerjaan diketahui). Mari kita tangani asumsi-asumsi tersebut selanjutnya.

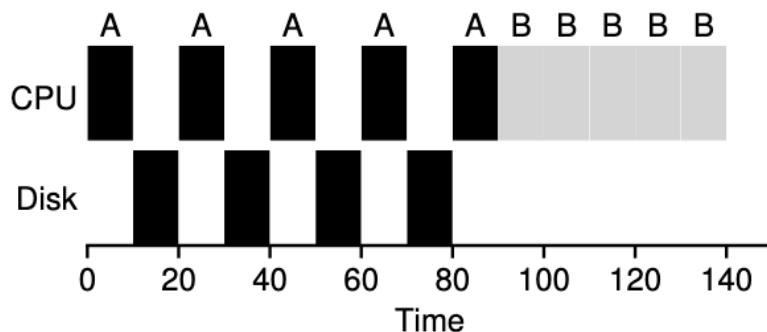
## 4.8 Menggabungkan I/O

Pertama kita akan melonggarkan asumsi 4 — tentu saja semua program melakukan I/O. Bayangkan program yang tidak menerima input apa pun: ia akan menghasilkan output yang sama setiap kali. Bayangkan program tanpa output: ia adalah pepatah pohon yang jatuh di hutan, tanpa ada yang melihatnya; tidak masalah bahwa ia berjalan.

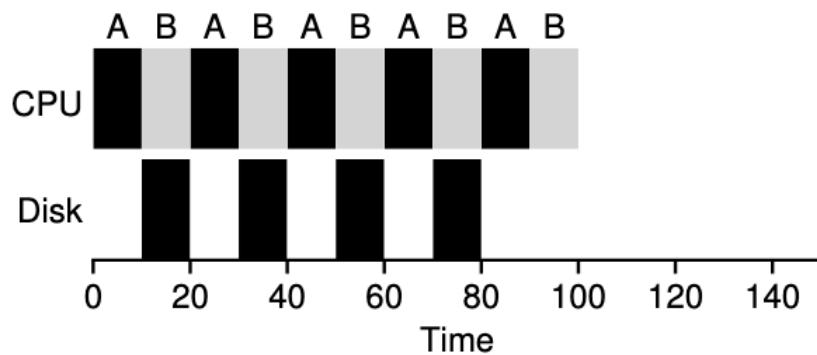
Seorang penjadwal jelas memiliki keputusan yang harus dibuat ketika sebuah pekerjaan memulai permintaan I/O, karena pekerjaan yang sedang berjalan tidak akan menggunakan CPU selama I/O; ia **terblokir** menunggu penyelesaian I/O. Jika I/O dikirim ke hard disk drive, proses mungkin terblokir selama beberapa milidetik atau lebih, tergantung pada beban I/O drive saat ini. Jadi, penjadwal mungkin harus menjadwalkan pekerjaan lain pada CPU pada saat itu.

Penjadwal juga harus membuat keputusan ketika I/O selesai. Ketika itu terjadi, sebuah interupsi dimunculkan, dan OS berjalan dan memindahkan proses yang mengeluarkan I/O dari terblokir kembali ke keadaan siap. Tentu saja, ia bahkan bisa memutuskan untuk menjalankan pekerjaan tersebut pada saat itu.

Untuk memahami masalah ini lebih baik, mari kita asumsikan kita memiliki dua pekerjaan, A dan B, yang masing-masing membutuhkan 50 ms waktu CPU. Namun, ada satu perbedaan yang jelas: A berjalan selama 10 ms dan kemudian mengeluarkan permintaan I/O (asumsikan di sini bahwa setiap I/O membutuhkan 10 ms), sedangkan B hanya menggunakan CPU selama 50 ms dan tidak melakukan I/O.



Gambar 4.8: Penggunaan Sumber Daya yang Buruk



Gambar 4.9: Overlap Memungkinkan Penggunaan Sumber Daya yang Lebih Baik

Pendekatan umum adalah memperlakukan setiap sub-pekerjaan 10 ms dari A sebagai pekerjaan independen. Jadi, ketika sistem dimulai, pilihannya adalah apakah akan menjadwalkan A 10 ms atau B 50 ms. Dengan STCF, pilihannya jelas: pilih yang lebih pendek, dalam kasus ini A. Kemudian, ketika sub-pekerjaan pertama A selesai, hanya B yang tersisa, dan ia mulai berjalan. Kemudian sub-pekerjaan baru A diajukan, dan ia mendahului B dan berjalan selama 10 ms. Melakukan hal ini memungkinkan **tumpang tindih** (*overlap*), dengan CPU digunakan oleh satu proses sementara menunggu I/O proses lain selesai; sistem dengan demikian digunakan dengan lebih baik (lihat Gambar 4.9).

### TIP: OVERLAP MEMUNGKINKAN PEMANFAATAN LEBIH TINGGI

Ketika memungkinkan, tumpang tindihkan operasi untuk memaksimalkan pemanfaatan sistem. Overlap berguna di banyak domain yang berbeda, termasuk ketika melakukan disk I/O atau mengirim pesan ke mesin jarak jauh; dalam kedua kasus tersebut, memulai operasi dan kemudian beralih ke pekerjaan lain adalah ide yang bagus, dan meningkatkan pemanfaatan dan efisiensi keseluruhan sistem.

Dan dengan demikian kita melihat bagaimana penjadwal mungkin menggabungkan I/O. Dengan memperlakukan setiap *CPU burst* sebagai pekerjaan, penjadwal memastikan proses-proses yang “interaktif” dijalankan secara sering. Sementara pekerjaan-pekerjaan interaktif tersebut melakukan I/O, pekerjaan CPU-intensif lainnya berjalan, sehingga memanfaatkan prosesor dengan lebih baik.

## 4.9 Tidak Lagi Maha Tahu

Dengan pendekatan dasar terhadap I/O sudah ada, kita sampai pada asumsi terakhir kita: bahwa penjadwal mengetahui panjang setiap pekerjaan. Seperti yang kita katakan sebelumnya, ini kemungkinan adalah asumsi terburuk yang bisa kita buat. Faktanya, dalam OS serbaguna (seperti yang kita pedulikan), OS biasanya tahu sangat sedikit tentang panjang setiap pekerjaan. Jadi, bagaimana kita dapat membangun pendekatan yang berperilaku seperti SJF/STCF tanpa pengetahuan *a priori* seperti itu? Lebih lanjut, bagaimana kita dapat menggabungkan beberapa ide yang telah kita lihat dengan penjadwal RR sehingga response time juga cukup baik?

## 4.10 Ringkasan

Kita telah memperkenalkan ide-ide dasar di balik penjadwalan dan mengembangkan dua keluarga pendekatan. Yang pertama menjalankan pekerjaan terpendek yang tersisa dan dengan demikian mengoptimalkan turnaround time; yang kedua bergantian antara semua pekerjaan dan dengan demikian mengoptimalkan response time. Keduanya buruk di tempat yang lain bagus, sayangnya, sebuah *trade-off* yang melekat yang umum dalam sistem. Kita juga telah melihat bagaimana kita mungkin menggabungkan I/O ke dalam gambaran, tetapi masih belum memecahkan masalah ketidakmampuan mendasar OS untuk melihat ke masa depan. Segera, kita akan melihat bagaimana mengatasi masalah ini, dengan membangun penjadwal yang menggunakan masa lalu terkini untuk memprediksi masa depan. Penjadwal ini dikenal

sebagai **multi-level feedback queue**, dan merupakan topik bab berikutnya.

## Referensi

- [B+79] "The Convoy Phenomenon" oleh M. Blasgen, J. Gray, M. Mitoma, T. Price. ACM Operating Systems Review, 13:2, April 1979.
- [C54] "Priority Assignment in Waiting Line Problems" oleh A. Cobham. Journal of Operations Research, 2:70, hlm. 70–76, 1954.
- [K64] "Analysis of a Time-Shared Processor" oleh Leonard Kleinrock. Naval Research Logistics Quarterly, 11:1, hlm. 59–73, Maret 1964.
- [CK68] "Computer Scheduling Methods and their Countermeasures" oleh Edward G. Coffman dan Leonard Kleinrock. AFIPS '68 (Spring), April 1968.
- [J91] "The Art of Computer Systems Performance Analysis" oleh R. Jain. Interscience, New York, April 1991.
- [O45] "Animal Farm" oleh George Orwell. Secker and Warburg (London), 1945.
- [PV56] "Machine Repair as a Priority Waiting-Line Problem" oleh Thomas E. Phipps Jr., W. R. Van Voorhis. Operations Research, 4:1, hlm. 76–86, Februari 1956.
- [MB91] "The effect of context switches on cache performance" oleh Jeffrey C. Mogul, Anita Borg. ASPLOS, 1991.

## Pekerjaan Rumah (Simulasi)

Program `scheduler.py` memungkinkan Anda melihat bagaimana penjadwal yang berbeda berkinerja di bawah metrik penjadwalan seperti response time, turnaround time, dan total waktu tunggu. Lihat README untuk detailnya.

### Pertanyaan

1. Hitung response time dan turnaround time saat menjalankan tiga pekerjaan dengan panjang 200 menggunakan penjadwal SJF dan FIFO.
2. Sekarang lakukan hal yang sama tetapi dengan pekerjaan yang panjangnya berbeda: 100, 200, dan 300.
3. Sekarang lakukan hal yang sama, tetapi juga dengan penjadwal RR dan irisan waktu 1.
4. Untuk jenis beban kerja apa SJF memberikan turnaround time yang sama dengan FIFO?
5. Untuk jenis beban kerja dan panjang kuantum apa SJF memberikan response time yang sama dengan RR?
6. Apa yang terjadi pada response time dengan SJF seiring panjang pekerjaan meningkat? Bisakah Anda menggunakan simulator untuk mendemonstrasikan tren tersebut?
7. Apa yang terjadi pada response time dengan RR seiring panjang kuantum meningkat? Bisakah Anda menulis persamaan yang memberikan response time kasus terburuk, diberikan  $N$  pekerjaan?



# Penjadwalan: Multi-Level Feedback Queue

Dalam bab ini, kita akan menangani masalah pengembangan salah satu pendekatan penjadwalan yang paling terkenal, yang dikenal sebagai **Multi-level Feedback Queue (MLFQ)**. Penjadwal Multi-level Feedback Queue (MLFQ) pertama kali dijelaskan oleh Corbato dkk. dalam sebuah sistem yang dikenal sebagai Compatible Time-Sharing System (CTSS) pada tahun 1962 [C+62], dan karya ini, bersama dengan karya selanjutnya pada Multics, membuat ACM menganugerahkan Corbato penghargaan tertingginya, yaitu Turing Award. Penjadwal ini kemudian disempurnakan selama bertahun-tahun hingga implementasi yang akan Anda temui di beberapa sistem modern.

Masalah mendasar yang coba ditangani MLFQ ada dua. Pertama, ia ingin mengoptimalkan **turnaround time**, yang, seperti kita lihat di bab sebelumnya, dilakukan dengan menjalankan pekerjaan yang lebih pendek lebih dulu; sayangnya, OS umumnya tidak tahu berapa lama pekerjaan akan berjalan, persis pengetahuan yang dibutuhkan oleh algoritma seperti SJF (atau STCF). Kedua, MLFQ ingin membuat sistem terasa responsif bagi pengguna interaktif (yaitu, pengguna yang duduk dan menatap layar, menunggu proses selesai), dan dengan demikian meminimalkan **response time**; sayangnya, algoritma seperti Round Robin mengurangi response time tetapi mengerikan untuk turnaround time.

Jadi, masalah kita: mengingat bahwa kita secara umum tidak mengetahui apa pun tentang suatu proses, bagaimana kita bisa membangun penjadwal untuk mencapai tujuan-tujuan ini? Bagaimana penjadwal bisa belajar, seiring sistem berjalan, tentang karakteristik pekerjaan yang dijalankannya, dan dengan demikian membuat keputusan penjadwalan yang lebih baik?

## INTI MASALAH: BAGAIMANA MENJADWALKAN TANPA PENGETAHUAN SEMPURNA?

Bagaimana kita bisa merancang penjadwal yang meminimalkan response time untuk pekerjaan interaktif sekaligus meminimalkan turnaround time tanpa pengetahuan *a priori* tentang panjang pekerjaan?

## TIP: BELAJAR DARI SEJARAH

Multi-level feedback queue adalah contoh yang sangat baik dari sistem yang belajar dari masa lalu untuk memprediksi masa depan. Pendekatan semacam itu umum dalam sistem operasi (dan banyak tempat lain dalam Ilmu Komputer, termasuk *hardware branch predictor* dan algoritma *caching*). Pendekatan semacam itu bekerja ketika pekerjaan memiliki fase perilaku dan dengan demikian dapat diprediksi; tentu saja, seseorang harus berhati-hati dengan teknik semacam itu, karena dengan mudah bisa salah dan mendorong sistem untuk membuat keputusan yang lebih buruk daripada yang akan dibuat tanpa pengetahuan sama sekali.

## 5.1 MLFQ: Aturan Dasar

Untuk membangun penjadwal seperti itu, dalam bab ini kita akan menjelaskan algoritma dasar di balik multi-level feedback queue; meskipun spesifikasi dari banyak MLFQ yang diimplementasikan berbeda [E95], sebagian besar pendekatannya serupa.

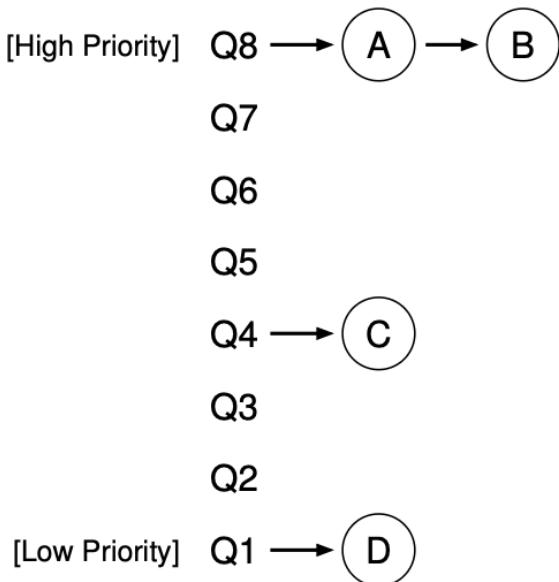
Dalam pembahasan kita, MLFQ memiliki sejumlah **antrian** yang berbeda, masing-masing diberi tingkat **prioritas** yang berbeda. Pada waktu tertentu, sebuah pekerjaan yang siap dijalankan berada pada satu antrian. MLFQ menggunakan prioritas untuk memutuskan pekerjaan mana yang harus dijalankan pada waktu tertentu: pekerjaan dengan prioritas lebih tinggi (yaitu, pekerjaan pada antrian yang lebih tinggi) dipilih untuk dijalankan.

Tentu saja, lebih dari satu pekerjaan mungkin berada pada antrian tertentu, dan dengan demikian memiliki prioritas yang sama. Dalam kasus ini, kita akan menggunakan penjadwalan round-robin di antara pekerjaan-pekerjaan tersebut. Dengan demikian, kita sampai pada dua aturan dasar pertama untuk MLFQ:

- **Aturan 1:** Jika  $\text{Priority}(A) > \text{Priority}(B)$ , A berjalan (B tidak).
- **Aturan 2:** Jika  $\text{Priority}(A) = \text{Priority}(B)$ , A & B berjalan dalam mode round-robin.

Kunci dari penjadwalan MLFQ oleh karena itu terletak pada bagaimana penjadwalan **mengatur prioritas**. Alih-alih memberikan prioritas tetap untuk setiap pekerjaan, MLFQ **memvariasikan** prioritas pekerjaan berdasarkan perilaku yang diamati. Jika, misalnya, sebuah pekerjaan berulang kali melepaskan CPU sementara menunggu input dari keyboard, MLFQ akan menjaga prioritasnya tetap tinggi, karena ini adalah perilaku proses interaktif. Jika sebaliknya, sebuah pekerjaan menggunakan CPU secara intensif untuk waktu yang lama, MLFQ akan mengurangi prioritasnya. Dengan cara ini, MLFQ akan mencoba **mempelajari** proses-proses saat mereka berjalan, dan dengan demikian menggunakan riwayat pekerjaan untuk memprediksi perilaku masa depannya.

Jika kita menggambarkan bagaimana antrian-antrian mungkin terlihat pada saat tertentu, kita mungkin melihat sesuatu seperti ini: dua pekerjaan (A dan B) berada pada tingkat prioritas tertinggi, sementara pekerjaan C berada di tengah dan Pekerjaan D berada pada prioritas terendah (Gambar 5.1). Dengan pengetahuan kita saat ini tentang bagaimana MLFQ bekerja, penjadwal hanya akan bergantian irisan waktu antara A dan B karena mereka adalah pekerjaan dengan prioritas tertinggi dalam sistem; pekerjaan malang C dan D tidak akan pernah mendapat kesempatan berjalan — sebuah ketidakadilan!



Gambar 5.1: Contoh MLFQ

Tentu saja, sekadar menunjukkan gambaran statis dari beberapa antrian tidak benar-benar memberi Anda ide tentang bagaimana MLFQ bekerja. Yang kita butuhkan adalah memahami bagaimana prioritas pekerjaan berubah dari waktu ke waktu.

## 5.2 Percobaan #1: Cara Mengubah Prioritas

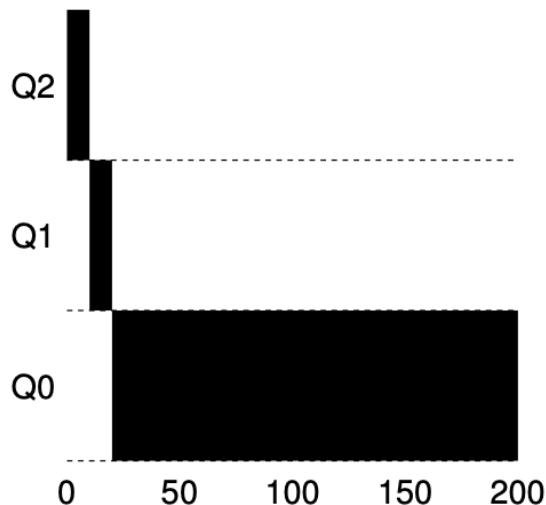
Sekarang kita harus memutuskan bagaimana MLFQ akan mengubah tingkat prioritas pekerjaan (dan dengan demikian antrian mana ia berada) selama masa hidup pekerjaan. Untuk melakukan ini, kita harus mengingat beban kerja kita: campuran pekerjaan interaktif yang berjalan singkat (dan mungkin sering melepaskan CPU), dan beberapa pekerjaan “CPU-bound” yang berjalan lebih lama yang membutuhkan banyak waktu CPU tetapi di mana response time tidak penting.

Untuk ini, kita membutuhkan konsep baru, yang akan kita sebut **jatah waktu** (*allotment*) pekerjaan. Jatah waktu adalah jumlah waktu yang dapat dihabiskan pekerjaan pada tingkat prioritas tertentu sebelum penjadwal mengurangi prioritasnya. Untuk kemudahan, awalnya kita akan mengasumsikan jatah waktu sama dengan satu irisan waktu. Berikut adalah percobaan pertama kita pada algoritma penyesuaian prioritas:

- **Aturan 3:** Ketika sebuah pekerjaan memasuki sistem, ia ditempatkan pada prioritas tertinggi (antrian paling atas).
- **Aturan 4a:** Jika pekerjaan menggunakan seluruh jatah waktunya saat berjalan, prioritasnya dikurangi (yaitu, ia turun satu antrian).
- **Aturan 4b:** Jika pekerjaan melepaskan CPU (misalnya, dengan melakukan operasi I/O) sebelum jatah waktu habis, ia tetap pada tingkat prioritas yang sama (yaitu, jatah waktunya direset).

## Contoh 1: Satu Pekerjaan Berjalan Lama

Mari kita lihat beberapa contoh. Pertama, kita akan melihat apa yang terjadi ketika ada pekerjaan yang berjalan lama dalam sistem, dengan irisan waktu 10 ms (dan dengan jatah waktu diatur sama dengan irisan waktu) pada penjadwal tiga antrian.

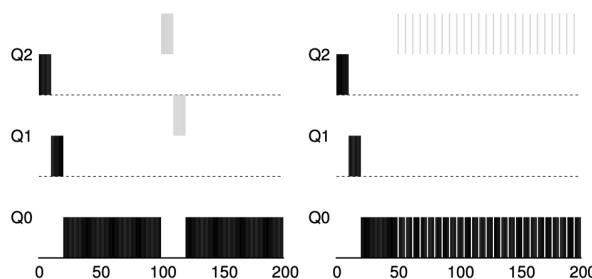


Gambar 5.2: Pekerjaan Berjalan Lama Dari Waktu ke Waktu

Seperti yang Anda lihat dalam contoh (Gambar 5.2), pekerjaan masuk pada prioritas tertinggi (Q2). Setelah satu irisan waktu 10 ms, penjadwal mengurangi prioritas pekerjaan satu tingkat, dan dengan demikian pekerjaan berada di Q1. Setelah berjalan di Q1 selama satu irisan waktu, pekerjaan akhirnya diturunkan ke prioritas terendah dalam sistem (Q0), di mana ia bertahan. Cukup sederhana, bukan?

## Contoh 2: Datangkan Pekerjaan Pendek

Sekarang mari kita lihat contoh yang lebih rumit, dan semoga melihat bagaimana MLFQ mencoba mendekati SJF. Dalam contoh ini, ada dua pekerjaan: A, yang merupakan pekerjaan CPU-intensif yang berjalan lama, dan B, yang merupakan pekerjaan interaktif yang berjalan singkat. Asumsikan A telah berjalan selama beberapa waktu, dan kemudian B tiba.



Gambar 5.3: Datangkan Pekerjaan Pendek

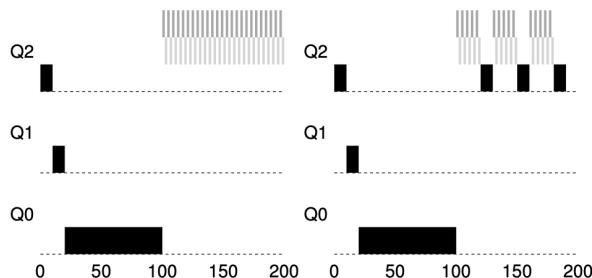
Pekerjaan A (ditunjukkan dengan hitam) berjalan di antrian prioritas terendah (seperti pekerjaan CPU-intensif yang berjalan lama lainnya); B (ditunjukkan dengan abu-abu) tiba pada

waktu  $T = 100$  dan dengan demikian dimasukkan ke antrian tertinggi; karena waktu eksekusinya singkat (hanya 20 ms), B selesai sebelum mencapai antrian terbawah, dalam dua irisan waktu; kemudian A melanjutkan berjalan (pada prioritas rendah) (lihat Gambar 5.3).

Dari contoh ini, Anda semoga bisa memahami salah satu tujuan utama algoritma: karena ia tidak tahu apakah sebuah pekerjaan akan menjadi pekerjaan pendek atau pekerjaan yang berjalan lama, ia pertama-tama mengasumsikan mungkin pekerjaan pendek, sehingga memberikan pekerjaan itu prioritas tinggi. Jika memang pekerjaan pendek, ia akan berjalan cepat dan selesai; jika bukan pekerjaan pendek, ia akan perlahan turun di antrian, dan dengan demikian segera membuktikan dirinya sebagai proses *batch* yang berjalan lama. Dengan cara ini, MLFQ mendekati SJF.

### Contoh 3: Bagaimana dengan I/O?

Sekarang mari kita lihat contoh dengan beberapa I/O. Seperti yang dinyatakan Aturan 4b di atas, jika sebuah proses melepaskan prosesor sebelum menggunakan jatah waktunya, kita menjaganya pada tingkat prioritas yang sama. Maksud dari aturan ini sederhana: jika pekerjaan interaktif, misalnya, melakukan banyak I/O (katakan dengan menunggu input pengguna dari keyboard atau mouse), ia akan melepaskan CPU sebelum jatah waktunya selesai; dalam kasus seperti itu, kita tidak ingin menghukum pekerjaan tersebut dan dengan demikian hanya menjaganya pada tingkat yang sama.



Gambar 5.4: Beban Kerja Campuran I/O dan CPU-intensif

Contoh ini menunjukkan pekerjaan interaktif B (ditunjukkan dengan abu-abu) yang membutuhkan CPU hanya selama 1 ms sebelum melakukan I/O, bersaing untuk CPU dengan pekerjaan *batch* yang berjalan lama A (ditunjukkan dengan hitam) (lihat Gambar 5.4). Pendekatan MLFQ menjaga B pada prioritas tertinggi karena B terus melepaskan CPU; jika B adalah pekerjaan interaktif, MLFQ selanjutnya mencapai tujuannya menjalankan pekerjaan interaktif dengan cepat.

### Masalah dengan MLFQ Kita Saat Ini

Kita dengan demikian memiliki MLFQ dasar. Tampaknya ia melakukan pekerjaan yang cukup baik, berbagi CPU secara adil di antara pekerjaan yang berjalan lama, dan membiarkan pekerjaan pendek atau I/O-intensif interaktif berjalan dengan cepat. Sayangnya, pendekatan yang telah kita kembangkan sejauh ini mengandung kelemahan serius. Bisakah Anda memikirkannya?

Pertama, ada masalah **kelaparan** (*starvation*): jika ada “terlalu banyak” pekerjaan interaktif dalam sistem, mereka akan bergabung untuk menghabiskan semua waktu CPU, dan dengan

demikian pekerjaan yang berjalan lama tidak akan pernah menerima waktu CPU (mereka kelaparan). Kita ingin membuat kemajuan pada pekerjaan-pekerjaan ini bahkan dalam skenario ini.

Kedua, pengguna yang cerdas dapat menulis ulang program mereka untuk **mempermainkan penjadwal** (*gaming the scheduler*). Mempermainkan penjadwal umumnya merujuk pada ide melakukan sesuatu licik untuk menipu penjadwal agar memberi Anda lebih dari bagian yang adil dari sumber daya. Algoritma yang telah kita jelaskan rentan terhadap serangan berikut: sebelum jatah waktu habis, keluarkan operasi I/O (misalnya, ke file) dan dengan demikian lepaskan CPU; melakukan hal itu memungkinkan Anda tetap di antrian yang sama, dan dengan demikian mendapatkan persentase waktu CPU yang lebih tinggi. Ketika dilakukan dengan benar (misalnya, dengan berjalan selama 99% dari jatah waktu sebelum melepaskan CPU), sebuah pekerjaan bisa hampir memonopoli CPU.

Akhirnya, sebuah program mungkin mengubah perilakunya dari waktu ke waktu; apa yang bersifat CPU-bound mungkin beralih ke fase interaktivitas. Dengan pendekatan kita saat ini, pekerjaan seperti itu akan kurang beruntung dan tidak diperlakukan seperti pekerjaan interaktif lainnya dalam sistem.

### TIP: PENJADWALAN HARUS AMAN DARI SERANGAN

Anda mungkin berpikir bahwa kebijakan penjadwalan, apakah di dalam OS itu sendiri (seperti yang dibahas di sini), atau dalam konteks yang lebih luas (misalnya, dalam pennanganan permintaan I/O sistem penyimpanan terdistribusi), bukan masalah keamanan, tetapi dalam semakin banyak kasus, ia memang persis itu. Pertimbangkan pusat data modern, di mana pengguna dari seluruh dunia berbagi CPU, memori, jaringan, dan sistem penyimpanan; tanpa kehati-hatian dalam desain dan penegakan kebijakan, satu pengguna mungkin dapat merugikan orang lain dan mendapatkan keuntungan untuk dirinya sendiri. Jadi, kebijakan penjadwalan membentuk bagian penting dari keamanan sistem, dan harus dibangun dengan hati-hati.

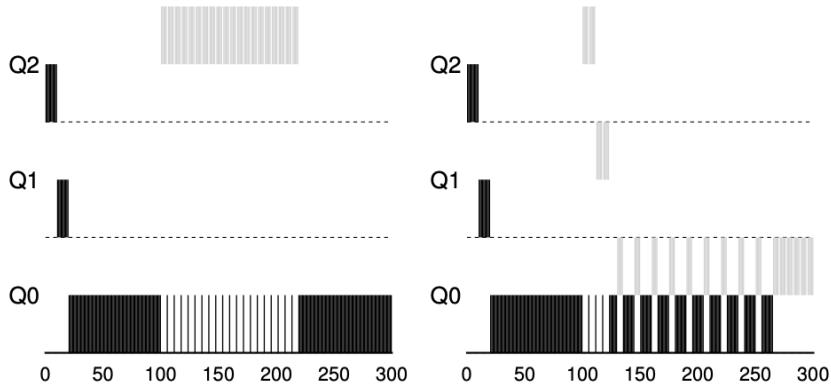
## 5.3 Percobaan #2: Priority Boost

Mari kita coba mengubah aturan dan lihat apakah kita bisa menghindari masalah kelaparan. Apa yang bisa kita lakukan untuk menjamin bahwa pekerjaan CPU-bound akan membuat kemajuan (meskipun tidak banyak)?

Ide sederhananya di sini adalah secara berkala **meningkatkan** (*boost*) prioritas semua pekerjaan dalam sistem. Ada banyak cara untuk mencapai ini, tetapi mari kita lakukan sesuatu yang sederhana: lempar semuanya ke antrian paling atas; oleh karena itu, sebuah aturan baru:

- **Aturan 5:** Setelah beberapa periode waktu  $S$ , pindahkan semua pekerjaan dalam sistem ke antrian paling atas.

Aturan baru kita memecahkan dua masalah sekaligus. Pertama, proses-proses dijamin tidak kelaparan: dengan duduk di antrian atas, sebuah pekerjaan akan berbagi CPU dengan pekerjaan prioritas tinggi lainnya dalam mode round-robin, dan dengan demikian akhirnya menerima layanan. Kedua, jika pekerjaan CPU-bound telah menjadi interaktif, penjadwal memperlakukannya dengan tepat setelah ia menerima peningkatan prioritas.



Gambar 5.5: Tanpa (Kiri) dan Dengan (Kanan) Priority Boost

Mari kita lihat sebuah contoh (Gambar 5.5). Dalam skenario ini, kita hanya menunjukkan perilaku pekerjaan yang berjalan lama ketika bersaing untuk CPU dengan dua pekerjaan interaktif yang berjalan singkat. Di sisi kiri, tidak ada peningkatan prioritas, dan dengan demikian pekerjaan yang berjalan lama kelaparan setelah dua pekerjaan pendek tiba; di sisi kanan, ada peningkatan prioritas setiap 100 ms (yang kemungkinan nilai yang terlalu kecil, tetapi digunakan di sini untuk contoh), dan dengan demikian kita setidaknya menjamin bahwa pekerjaan yang berjalan lama akan membuat kemajuan, dinaikkan ke prioritas tertinggi setiap 100 ms dan dengan demikian mendapat kesempatan berjalan secara berkala.

Tentu saja, penambahan periode waktu  $S$  mengarah pada pertanyaan yang jelas: berapa yang harus ditetapkan untuk  $S$ ? John Ousterhout, seorang peneliti sistem yang dihormati [O11], biasa menyebut nilai-nilai seperti itu dalam sistem sebagai **konstanta voo-doo**, karena tampaknya memerlukan semacam sihir hitam untuk mengaturnya dengan benar. Sayangnya,  $S$  memiliki nuansa itu. Jika diatur terlalu tinggi, pekerjaan yang berjalan lama bisa kelaparan; terlalu rendah, dan pekerjaan interaktif mungkin tidak mendapatkan bagian CPU yang tepat.

## 5.4 Percobaan #3: Akuntansi yang Lebih Baik

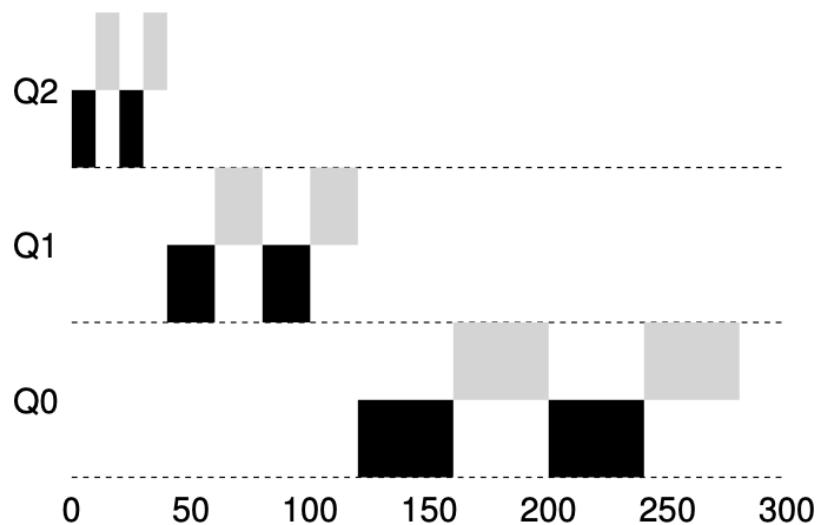
Sekarang kita memiliki satu masalah lagi untuk dipecahkan: bagaimana mencegah pemermainan penjadwal kita? Pelaku sebenarnya di sini, seperti yang mungkin Anda duga, adalah Aturan 4a dan 4b, yang memungkinkan pekerjaan mempertahankan prioritasnya dengan melepaskan CPU sebelum jatah waktunya habis. Jadi apa yang harus kita lakukan?

## TIP: HINDARI KONSTANTA VOO-DOO (HUKUM OUSTERHOUT)

Menghindari konstanta voo-doo adalah ide yang baik kapan pun memungkinkan. Sayangnya, seperti dalam contoh di atas, sering kali sulit. Seseorang bisa mencoba membuat sistem belajar nilai yang baik, tetapi itu juga tidak mudah. Hasil yang sering terjadi: file konfigurasi yang dipenuhi dengan nilai parameter default yang bisa disesuaikan oleh administrator berpengalaman ketika sesuatu tidak berjalan dengan benar. Seperti yang bisa Anda bayangkan, ini sering dibiarkan tidak dimodifikasi, dan dengan demikian kita dibiarkan berharap bahwa default bekerja dengan baik di lapangan. Tip ini dibawakan oleh profesor OS lama kita, John Ousterhout, dan karenanya kita menyebutnya Hukum Ousterhout.

Solusi di sini adalah melakukan **akuntansi yang lebih baik** terhadap waktu CPU pada setiap tingkat MLFQ. Alih-alih lupakan berapa banyak jatah waktu yang telah digunakan proses pada tingkat tertentu ketika ia melakukan I/O, penjadwal harus melacaknya; setelah proses telah menggunakan jatah waktunya, ia diturunkan ke antrian prioritas berikutnya. Apakah ia menggunakan jatah waktunya dalam satu *burst* panjang atau banyak yang pendek seharusnya tidak penting. Kita dengan demikian menulis ulang Aturan 4a dan 4b menjadi satu aturan berikut:

- **Aturan 4:** Setelah pekerjaan menggunakan jatah waktunya pada tingkat tertentu (terlepas dari berapa kali ia telah melepaskan CPU), prioritasnya dikurangi (yaitu, ia turun satu antrian).



Gambar 5.6: Tanpa (Kiri) dan Dengan (Kanan) Toleransi Pemermainan

Mari kita lihat sebuah contoh (Gambar 5.6). Di sisi kiri, tanpa perlindungan dari pemermainan, sebuah proses dapat mengeluarkan I/O sebelum jatah waktunya habis, sehingga tetap pada tingkat prioritas yang sama, dan mendominasi waktu CPU. Dengan akuntansi yang lebih baik (di sisi kanan), terlepas dari perilaku I/O proses, ia perlahan turun di antrian, dan dengan demikian tidak dapat memperoleh bagian CPU yang tidak adil.

## 5.5 Tuning MLFQ dan Isu Lainnya

Beberapa isu lain muncul dengan penjadwalan MLFQ. Satu pertanyaan besar adalah bagaimana memparameterisasi penjadwal seperti itu. Misalnya, berapa banyak antrian yang harus ada? Berapa besar irisan waktu per antrian? Jatah waktunya? Seberapa sering prioritas harus dinaikkan untuk menghindari kelaparan dan memperhitungkan perubahan perilaku? Tidak ada jawaban mudah untuk pertanyaan-pertanyaan ini, dan dengan demikian hanya pengalaman dengan beban kerja dan tuning selanjutnya dari penjadwal yang akan mengarah pada keseimbangan yang memuaskan.

Misalnya, sebagian besar varian MLFQ memungkinkan panjang irisan waktu yang bervariasi di antara antrian yang berbeda. Antrian prioritas tinggi biasanya diberi irisan waktu pendek; mereka terdiri dari pekerjaan interaktif, bagaimanapun juga, dan dengan demikian bergantian dengan cepat di antara mereka masuk akal (misalnya, 10 ms atau kurang). Antrian prioritas rendah, sebaliknya, berisi pekerjaan yang berjalan lama yang bersifat CPU-bound; oleh karena itu, irisan waktu yang lebih panjang bekerja dengan baik (misalnya, 100-an ms).

Implementasi MLFQ Solaris — kelas penjadwalan Time-Sharing, atau TS — sangat mudah dikonfigurasi; ia menyediakan sekumpulan tabel yang menentukan persis bagaimana prioritas suatu proses diubah sepanjang masa hidupnya, berapa lama setiap irisan waktu, dan seberapa sering untuk meningkatkan prioritas pekerjaan [AD00]; seorang administrator dapat mengutak-atik tabel ini untuk membuat penjadwal berperilaku dengan cara yang berbeda. Nilai default untuk tabel adalah 60 antrian, dengan panjang irisan waktu yang meningkat perlahan dari 20 milidetik (prioritas tertinggi) hingga beberapa ratus milidetik (terendah), dan prioritas dinaikkan sekitar setiap 1 detik atau lebih.

Penjadwal MLFQ lainnya tidak menggunakan tabel atau aturan persis yang dijelaskan dalam bab ini; melainkan mereka menyesuaikan prioritas menggunakan formula matematika. Misalnya, penjadwal FreeBSD (versi 4.3) menggunakan formula untuk menghitung tingkat prioritas saat ini dari sebuah pekerjaan, mendasarkannya pada berapa banyak CPU yang telah digunakan proses [LM+89]; selain itu, penggunaan meluruh dari waktu ke waktu, memberikan peningkatan prioritas yang diinginkan dengan cara yang berbeda dari yang dijelaskan di sini. Lihat makalah Epema untuk tinjauan yang sangat baik tentang algoritma *decay-usage* dan propertiinya [E95].

Akhirnya, banyak penjadwal memiliki beberapa fitur lain yang mungkin Anda temui. Misalnya, beberapa penjadwal memesan tingkat prioritas tertinggi untuk pekerjaan sistem operasi; dengan demikian pekerjaan pengguna biasa tidak pernah bisa mendapatkan tingkat prioritas tertinggi dalam sistem. Beberapa sistem juga memungkinkan beberapa saran pengguna untuk membantu mengatur prioritas; misalnya, dengan menggunakan utilitas baris perintah nice Anda dapat meningkatkan atau mengurangi prioritas pekerjaan (sedikit) dan dengan demikian meningkatkan atau mengurangi peluangnya untuk berjalan pada waktu tertentu.

## TIP: GUNAKAN SARAN JIKA MEMUNGKINKAN

Karena sistem operasi jarang tahu apa yang terbaik untuk setiap proses di sistem, seiring kali berguna untuk menyediakan antarmuka yang memungkinkan pengguna atau administrator memberikan beberapa petunjuk ke OS. Kita sering menyebut petunjuk tersebut sebagai **saran** (*advice*), karena OS tidak selalu harus memperhatikannya, tetapi mungkin memperhitungkan saran tersebut untuk membuat keputusan yang lebih baik. Petunjuk semacam itu berguna di banyak bagian OS, termasuk penjadwal (misalnya, dengan `nice`), manajer memori (misalnya, `madvise`), dan sistem file (misalnya, *informed prefetching and caching*).

## 5.6 MLFQ: Ringkasan

Kita telah menjelaskan pendekatan penjadwalan yang dikenal sebagai **Multi-Level Feedback Queue (MLFQ)**. Semoga Anda sekarang bisa melihat mengapa ia dinamai seperti itu: ia memiliki banyak tingkat antrian, dan menggunakan umpan balik (*feedback*) untuk menentukan prioritas pekerjaan tertentu. Sejarah adalah panduannya: perhatikan bagaimana pekerjaan berperilaku dari waktu ke waktu dan perlakukan mereka sesuai.

Kumpulan aturan MLFQ yang disempurnakan, yang tersebar di seluruh bab, direproduksi di sini:

- **Aturan 1:** Jika  $\text{Priority}(A) > \text{Priority}(B)$ , A berjalan (B tidak).
- **Aturan 2:** Jika  $\text{Priority}(A) = \text{Priority}(B)$ , A & B berjalan dalam mode round-robin menggunakan irisan waktu (panjang kuantum) dari antrian yang bersangkutan.
- **Aturan 3:** Ketika sebuah pekerjaan memasuki sistem, ia ditempatkan pada prioritas tertinggi (antrian paling atas).
- **Aturan 4:** Setelah pekerjaan menggunakan jatah waktunya pada tingkat tertentu (terlepas dari berapa kali ia telah melepaskan CPU), prioritasnya dikurangi (yaitu, ia turun satu antrian).
- **Aturan 5:** Setelah beberapa periode waktu  $S$ , pindahkan semua pekerjaan dalam sistem ke antrian paling atas.

MLFQ menarik karena alasan berikut: alih-alih menuntut pengetahuan *a priori* tentang sifat pekerjaan, ia mengamati eksekusi pekerjaan dan memprioritaskannya sesuai. Dengan cara ini, ia berhasil mencapai yang terbaik dari kedua dunia: ia dapat memberikan kinerja keseluruhan yang sangat baik (mirip dengan SJF/STCF) untuk pekerjaan interaktif yang berjalan singkat, dan bersifat adil serta membuat kemajuan untuk beban kerja CPU-intensif yang berjalan lama. Untuk alasan ini, banyak sistem, termasuk turunan BSD UNIX [LM+89, B86], Solaris [M06], dan Windows NT serta sistem operasi Windows berikutnya [CS97] menggunakan bentuk MLFQ sebagai penjadwal dasar mereka.

## Referensi

- [AD00] “Multilevel Feedback Queue Scheduling in Solaris” oleh Andrea Arpaci-Dusseau.

Tersedia: <http://www.ostep.org/Citations/notes-solaris.pdf>.

- [B86] "The Design of the UNIX Operating System" oleh M.J. Bach. Prentice-Hall, 1986.
- [C+62] "An Experimental Time-Sharing System" oleh F. J. Corbato, M. M. Daggett, R. C. Daley. IFIPS 1962.
- [CS97] "Inside Windows NT" oleh Helen Custer dan David A. Solomon. Microsoft Press, 1997.
- [E95] "An Analysis of Decay-Usage Scheduling in Multiprocessors" oleh D.H.J. Epema. SIGMETRICS '95.
- [LM+89] "The Design and Implementation of the 4.3BSD UNIX Operating System" oleh S.J. Leffler, M.K. McKusick, M.J. Karels, J.S. Quarterman. Addison-Wesley, 1989.
- [M06] "Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture" oleh Richard McDougall. Prentice-Hall, 2006.
- [O11] "John Ousterhout's Home Page" oleh John Ousterhout. [www.stanford.edu/~ouster/](http://www.stanford.edu/~ouster/).
- [P+95] "Informed Prefetching and Caching" oleh R.H. Patterson dkk. SOSP '95, Copper Mountain, Colorado, Oktober 1995.

## Pekerjaan Rumah (Simulasi)

Program `mlfq.py` memungkinkan Anda melihat bagaimana penjadwal MLFQ yang disajikan dalam bab ini berperilaku. Lihat README untuk detailnya.

### Pertanyaan

1. Jalankan beberapa masalah yang dihasilkan secara acak dengan hanya dua pekerjaan dan dua antrian; hitung jejak eksekusi MLFQ untuk masing-masing. Buat hidup Anda lebih mudah dengan membatasi panjang setiap pekerjaan dan menonaktifkan I/O.
2. Bagaimana Anda akan menjalankan penjadwal untuk mereproduksi setiap contoh dalam bab ini?
3. Bagaimana Anda akan mengkonfigurasi parameter penjadwal agar berperilaku persis seperti penjadwal round-robin?
4. Buatlah beban kerja dengan dua pekerjaan dan parameter penjadwal sehingga satu pekerjaan memanfaatkan Aturan 4a dan 4b lama (dinyalakan dengan flag `-s`) untuk mempermainkan penjadwal dan mendapatkan 99% CPU selama interval waktu tertentu.
5. Diberikan sistem dengan panjang kuantum 10 ms di antrian tertingginya, seberapa sering Anda harus menaikkan pekerjaan kembali ke tingkat prioritas tertinggi (dengan flag `-B`) untuk menjamin bahwa satu pekerjaan yang berjalan lama (dan berpotensi kelaparan) mendapatkan setidaknya 5% CPU?
6. Satu pertanyaan yang muncul dalam penjadwalan adalah ujung antrian mana untuk menambahkan pekerjaan yang baru selesai I/O; flag `-I` mengubah perilaku ini untuk simulator penjadwalan ini. Bermain-mainlah dengan beberapa beban kerja dan lihat apakah Anda bisa melihat efek dari flag ini.