



## MODUL PERKULIAHAN

# Sistem Operasi

## Pendahuluan

Fakultas

Ilmu Komputer

Program Studi

Teknik Informatika

Tatap Muka

**01**

Kode MK

87030

Disusun Oleh

Tim Dosen

### Abstract

Modul ini membahas tentang Dasar Sistem operasi dan system komputer

### Kompetensi

Diharapkan mahasiswa dapat memahami bagian-bagian dari sebuah system komputer

# Pendahuluan

## Defenisi umum

---

Sistem Operasi merupakan sebuah program yang mengelola perangkat keras computer. OS juga menyediakan sebuah basis untuk program aplikasi dan bertindak sebagai penghubung antara pengguna computer dan perangkat keras komputer.

## Sistem Komputer

---

Sebuah sistem computer dibagi menjadi 4 komponen (Gambar 1.1), yaitu :

1. Hardware

Hardware atau perangkat keras terdiri dari:

- Central Processing Unit (CPU)
- Memory
- Input/Output (I/O) Device

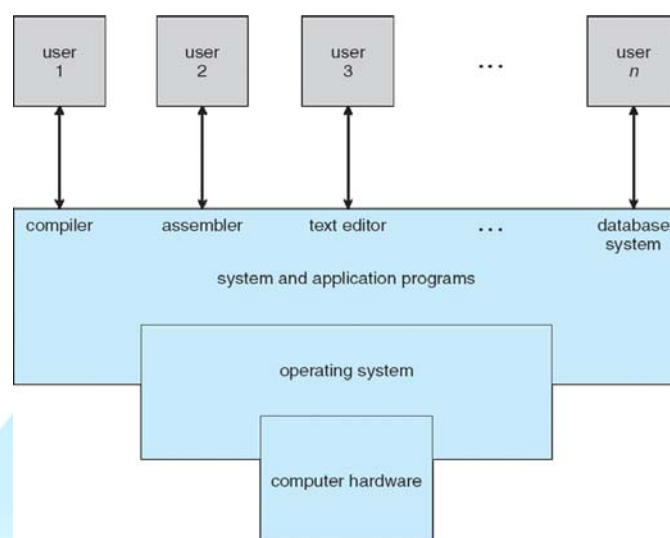
2. Sistem Operasi (OS)

OS mengontrol hardware dan berkoordinasi dengan berbagai program aplikasi dan berbagai user

3. Program Aplikasi

Sebuah program yang digunakan untuk menyelesaikan sebuah masalah, seperti word processor, spreadsheets, compiler, web browser, dll

4. User



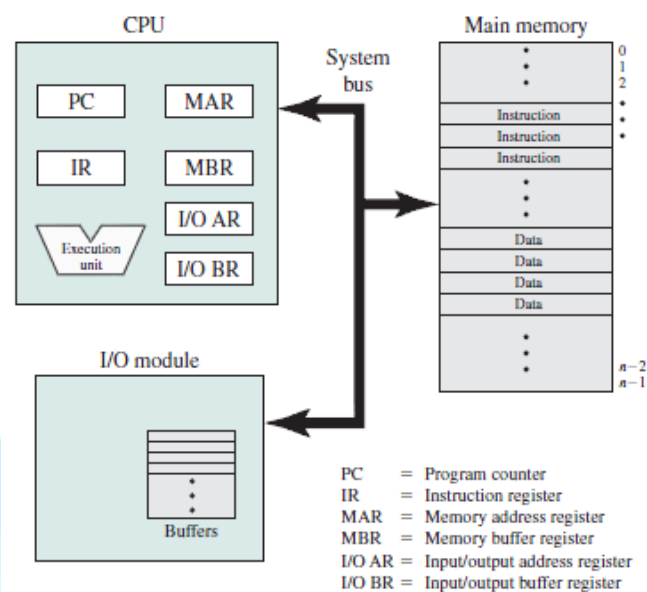
Gambar 1.1. Komponen Utama Sistem Komputer

# Sistem Komputer

## Elemen Dasar

Sebuah computer terdiri dari elemen dasar : processor, memory dan komponen I/O (Gambar 1.2) Komponen-komponen ini saling terhubung dengan cara yang sama untuk mencapai fungsi utama dari computer yaitu eksekusi program.

- **Processor**  
Melakukan control operasi dari computer dan melaksanakan fungsi-fungsi pemrosesan data. Processor sering disebut juga dengan Central Processing Unit (CPU).
- **Main Memory**  
Menyimpan data dan program. Memori ada yang bersifat volatile dan non volatile. Memory yang bersifat volatile, ketika computer mati, maka isi dari memory akan hilang, sedangkan yang non volatile, isi memori akan tetap ada, walaupun computer dimatikan. Main memori biasanya mengacu kepada RAM (Random Akses Memory) atau memory utama.
- **I/O Module**  
Data berpindah antara komputer dan lingkungan luar. Lingkungan luar ini terdiri dari beragam perangkat, termasuk memori sekunder seperti harddisk, peralatan komunikasi, dan terminal-terminal.
- **System Bus**  
Menyediakan komunikasi antara processor, main memory dan I/O module.



**Gambar 1.2. Elemen Dasar Komputer**

## Processor / Central Processing Unit

---

Processor adalah otaknya komputer, yang melakukan control operasi dari computer dan melaksanakan fungsi-fungsi pemrosesan data. Bagian dari processor adalah :

1. Arithmetic Logical Unit (ALU)

Sebuah unit yang berfungsi untuk melakukan operasi-operasi terkait dengan logika dan matematika.

2. Control Unit

Sebuah unit yang mengendalikan operasi computer, dan mengendalikan aliran data berdasarkan instruksi-instruksi.

3. Register

Merupakan sebuah media penyimpanan sementara bagi processor. Contoh-contoh register dasar pada CPU:

- Instruction Register (IR)  
Berfungsi menyimpan instruksi yang dijemput dari memori
- Program Counter (PC)  
Berfungsi untuk menyimpan alamat instruksi di memory yang akan dieksekusi oleh CPU
- Memori Address Register (MAR)  
Berisi alamat memori yang akan dibaca dan ditulis
- Memory Buffer Register (MBR)  
Berisi data yang akan ditulis ke memori atau menerima data dari memori
- I/O Address Register (I/O AR)  
Berisi alamat I/O module yang akan dibaca atau ditulis
- I/O Buffer Register (I/O AR)  
Digunakan untuk pertukaran data antara I/O module dan processor

## Eksekusi Instruksi

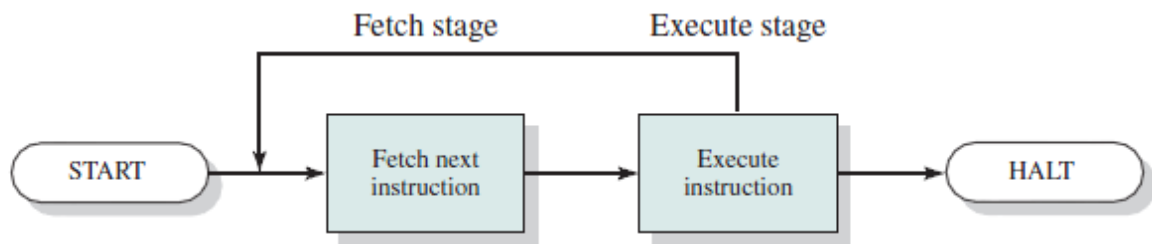
---

Sebuah program dieksekusi oleh processor terdiri dari serangkaian instruksi yang disimpan didalam memori. Untuk memproses setiap instruksi tersebut, adalah dengan 2 langkah :

- Processor membaca instruksi dari memori pada satu waktu → fetches
- Eksekusi instruksi → execute

Untuk mengeksekusi sebuah program, maka akan terjadi perulangan proses instruction fetch dan and instruction execution.

Pemrosesan yang membutuhkan sebuah single instruksi disebut sebagai sebuah instruction cycle (Gambar 1.3).

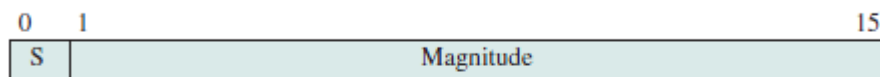


**Gambar 1.3 Dasar Instruction cycle**

Sebuah instruksi memiliki sebuah format (tergantung pada jenis mesin), yang terdiri dari 2 bagian utama yaitu opcode dan address. Opcode merupakan sebuah kode untuk operasi yang akan dilaksanakan, sedangkan address merupakan alamat memori dimana data disimpan. Sebagai contoh Gambar 1.4(a) merupakan format instruksi dari mesin Hypothetical.



(a) Instruction format



(b) Integer format

Program counter (PC) = Address of instruction  
 Instruction register (IR) = Instruction being executed  
 Accumulator (AC) = Temporary storage

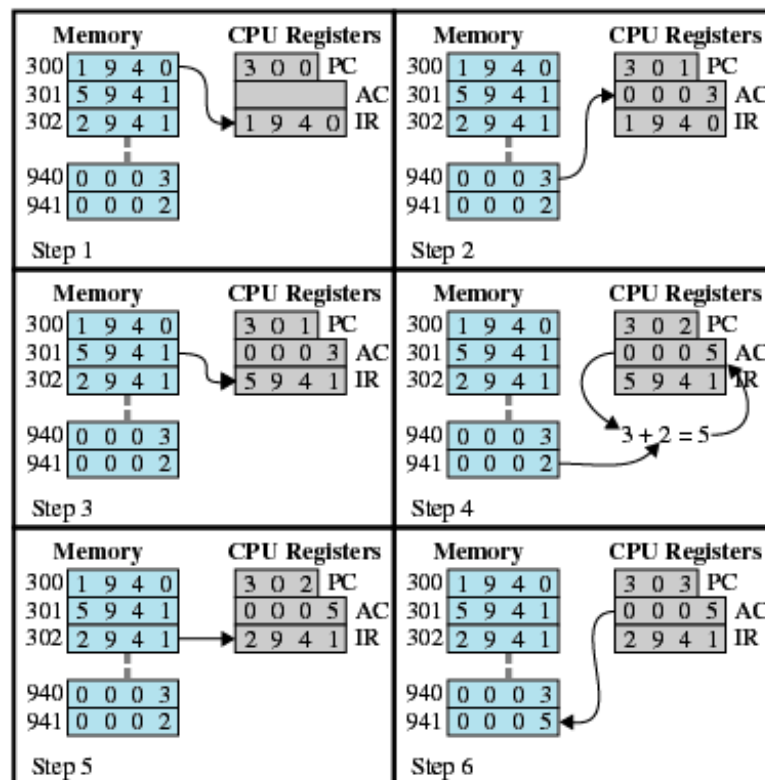
(c) Internal CPU registers

0001 = Load AC from memory  
 0010 = Store AC to memory  
 0101 = Add to AC from memory

(d) Partial list of opcodes

**Gambar 1.4. Contoh instruksi mesin Hypothetical**

Contoh eksekusi program dapat dilihat pada Gambar 1.5 berikut.



Gambar 1.5 Contoh eksekusi program

Pada awal setiap instruction cycle, processor akan menjemput (fetches) sebuah instruksi dari memori. Biasanya program counter (PC) menyimpan alamat dari instruksi berikutnya yang akan di-fetch. Processor akan selalu menambah satu nilai dari PC (increment), setelah satu instruksi selesai dijemput, sehingga untuk fetch instruksi berikutnya akan selalu berurutan.

## Contoh

Menggunakan Hypothetical processor. Processor ini terdiri dari beberapa register:

- Accumulator (AC), berfungsi sebagai tempat penyimpanan sementara
- Instruction Register (IR), berfungsi menyimpan instruksi yang akan dieksekusi
- Program Counter (PC), berfungsi menyimpan next instruction.

Setiap instruksinya dan data, memiliki panjang 16 bit

Instruksi format terdiri dari

- 4 bit Opcode (Operasional Code)
  - 0001 → load AC from memory
  - 0010 → Store AC to memory
  - 0101 → Add to AC from memory

Operasi yang terjadi adalah sebagai berikut (Gambar 1.5)

1. PC berisi nilai 300, merupakan alamat instruksi pertama. Instruksi ini bernilai 1940 (heksadesimal), diload kedalam IR dan nilai PC+1 (Increment) → 301
2. 1940 dalam IR terdiri dari opcode (1 atau 0001) dan alamat (940). Sehingga instruksi ini mengindikasikan bahwa AC akan di-load dengan data di memory pada alamat 940 (dalam contoh AC = 3).

**CTT>> Proses 1 & 2 adalah 1 cycle Instruction (proses 1 → fetch, proses 2 → execute)**

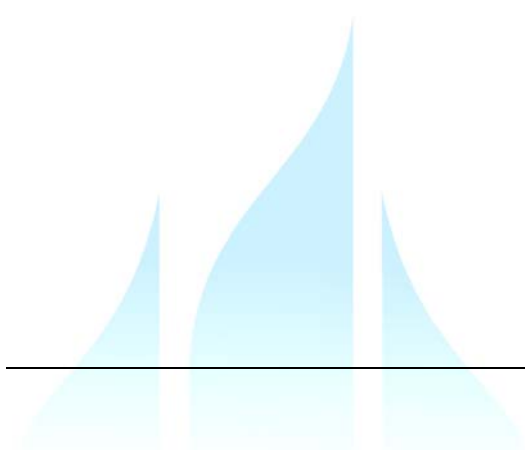
3. PC berisi nilai 301, merupakan alamat instruksi berikutnya. Instruksi ini bernilai 1941 (heksadesimal), diload kedalam IR dan nilai PC+1 (Increment) → 302
4. 5941 dalam IR terdiri dari opcode (5 atau 0101) dan alamat (941). Sehingga instruksi ini mengindikasikan bahwa isi lama dari AC (3) dan isi dari memori pada lokasi 941 → (2), dijumlahkan dan hasilnya disimpan kembali ke AC, sehingga AC=5

**CTT>> Proses 3 & 4 adalah 1 cycle Instruction (proses 3 → fetch, proses 4 → execute)**

5. PC berisi nilai 302, merupakan alamat instruksi berikutnya. Instruksi ini bernilai 2941 (heksadesimal), diload kedalam IR dan nilai PC+1 (Increment) → 303
6. 2941 dalam IR terdiri dari opcode (2 atau 0010) dan alamat (941). Sehingga instruksi ini mengindikasikan bahwa isi dari AC (5) di-store ke lokasi memori 941.

**CTT>> Proses 5 & 6 adalah 1 cycle Instruction (proses 5 → fetch, proses 6 → execute)**

Sehingga untuk contoh disini, dibutuhkan 3 cycle instruction untuk menyelesaikan penjumlahan  $2+3=5$

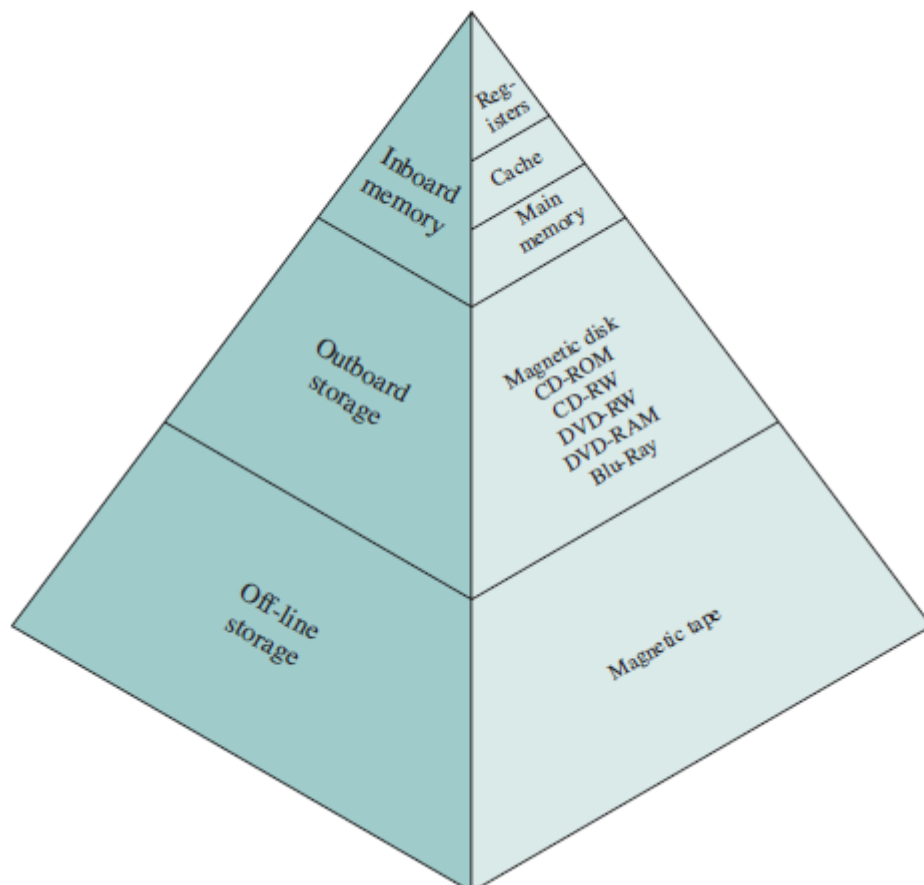


# Memory

## Hirarki Memory

Ada 3 karakteristik kunci dari memori, yaitu kapasitas, waktu akses dan biaya. Untuk ketiga karakteristik tersebut saling memiliki kaitan, Gambar 1.6 menyajikan kaitannya dalam sebuah hirarki memori, yaitu semakin kebawah maka:

- Penurunan biaya/bit
- Peningkatan kapasitas
- Peningkatan waktu akses
- Pengurangan frekuensi akses ke memory oleh processor.



Gambar 1.6 Hirarki Memori



# Struktur I/O

Sebagian besar dari kode OS ditujukan untuk menangani I/O. sebuah general purpose register terdiri dari CPU dan banyak device controller yang dihubungkan dengan sebuah bus. Setiap device controller, menangani 1 atau banyak jenis perangkat, tergantung pada controller. Sebagai contoh, 7 atau banyak perangkat, bisa ditancapkan ke small computer system interface (SCSI).

Sebuah device controller memiliki beberapa local buffer storage dan serangkaian special-purpose registers. Device controller bertanggung jawab untuk memindahkan data antara peripheral devices yang dikontrol dan local buffer storage. OS biasanya memiliki sebuah **device driver** untuk setiap device controller. Device driver ini memahami device controller dan menyediakan bagian dari operating system dengan sebuah interface seragam ke device.

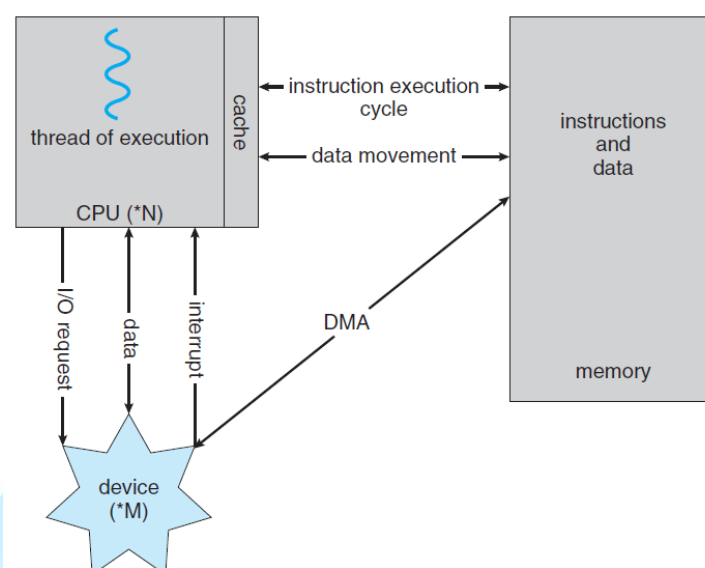
## Operasi I/O

### 1. Programmed I/O

Pada saat processor mengeksekusi sebuah program dan

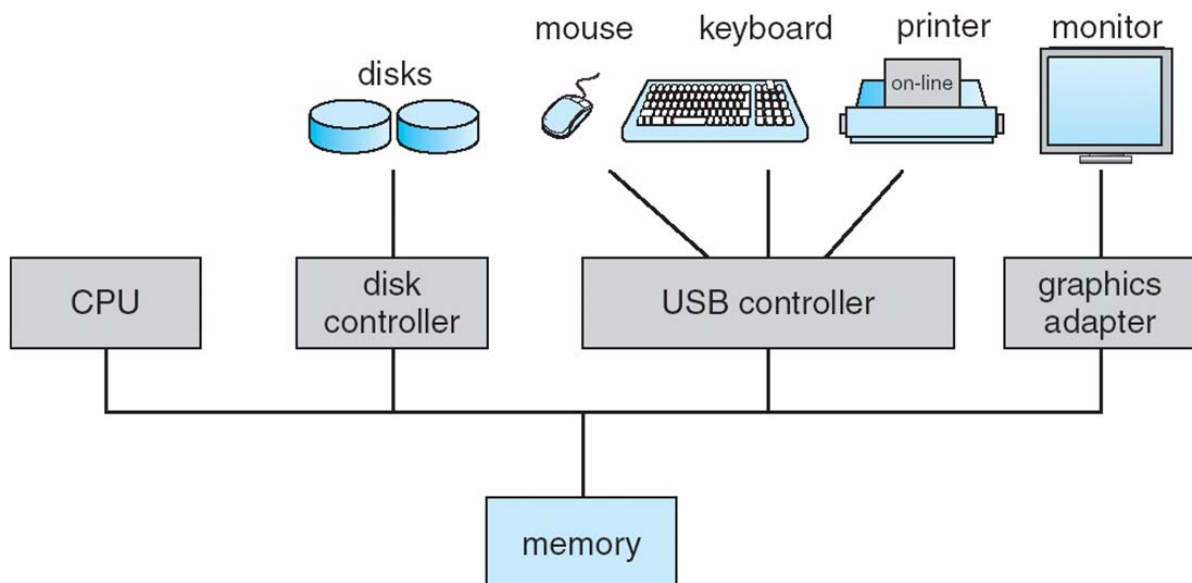
### 2. Interrupt-Driven I/O

### 3. Direct Memory Access (DMA)



# Operasi Sistem Komputer

Sebuah sistem komputer general-purpose modern terdiri dari satu atau banyak processor dan sejumlah device controller yang dihubungkan dengan sebuah common bus yang menyediakan akses ke *shared memory* (Gambar 1.6). setiap device controller akan melayani satu jenis perangkat (ex, disk drives, audio devices, or video displays).



Gambar 1.6 Sistem computer modern

CPU dan perangkat controllers dapat berjalan secara parallel, dan bersaing untuk memory cycles. Untuk menjamin urutan akses ke shared memory, maka sebuah memory controller melakukan sinkronisasi akses ke memory.

## Bootstrap Program

Sebuah komputer agar dapat running, maka diperlukan pada saat powered up / rebooted—diperlukan sebuah initial program untuk run. initial program, disebut dengan **bootstrap program**. Bootstrap program biasanya disimpan dalam sebuah perangkat keras computer yaitu :

- read-only memory (**ROM**)
- electrically erasable programmable read-only memory (**EEPROM**)

yang dikenal dengan istilah **firmware**.

Bootstrap program akan mengenali semua aspek-aspek sistem, dari CPU register sampai device controller dan isi memori. Bootstrap program juga harus mengetahui bagaimana cara untuk load OS dan memulai eksekusi sistem.

Untuk mencapai tujuan ini, maka bootstrap program harus menempatkan OS kernel dan me-loadnya ke dalam memori. Ketika kernel di-load dan di-eksekusi, maka kernel sudah dapat untuk mulai menyediakan layanan ke sistem dan user.

Beberapa layanan disediakan diluar kernel, oleh program sistem yang di-load kedalam memori pada saat boot time dan menjadi system processes atau system daemons, yang berjalan selama kernel berjalan.



# Daftar Pustaka

1. Stalling.W, *Operating System Internals and Design Principle Seventh Edition*, Prentice hall, 2012
2. Silberschatz. A, Galvin. P.B, Gagne. Greg, *Operating System Concepts Ninth perEdition*, Jhon, Wiley & Sons, 2013



## MODUL PERKULIAHAN

# Sistem Operasi

## Arsitektur Sistem Komputer & Struktur OS

Fakultas  
Ilmu Komputer

Program Studi  
Teknik Informatika

Tatap Muka

02

Kode MK  
87030

Disusun Oleh  
Tim Dosen

### Abstract

Modul ini membahas tentang  
Arsitektur Sistem Komputer & Struktur  
SIStem Operasi

### Kompetensi

Diharapkan mahasiswa dapat  
memahami arsitektur single dan  
multiprocessor, dan juga  
multiprogramming

# Arsitektur Sistem Komputer

## Sistem Single Processor

---

Sebagian besar system computer menggunakan sebuah processor tunggal. Pada processor tunggal, terdapat 1 CPU utama yang mampu melakukan eksekusi serangkaian instruksi general purpose, termasuk instruksi dari proses uses.

## Sistem Multiprocessor

---

Sistem multiprocessor dikenal juga dengan system parallel atau system multicore. System ini memiliki dua atau lebih processor dalam sebuah komunikasi tertutup, berbagi bus, , clock, memory, dan perangkat peripheral.

Multiprocessor memiliki 3 keuntungan utaman:

1. Meningkatkan throughput.

Dengan meningkatnya jumlah processor, maka akan menyebabkan pekerjaan yang dilakukan lebih cepat.

2. Lebih ekonomis

System Multiprocessor lebih hemat dibandingkan dengan system multiple single-processor system, karena multiprocessor dapat berbagi peripherals, media penyimpanan, dan juga power supplies.

Jika beberapa program beroperasi pada serangkaian data yang sa,a, maka lebih hemat untuk menyimpan data tersebut pada 1 harddisk.

3. Meningkatkan ketersediaan

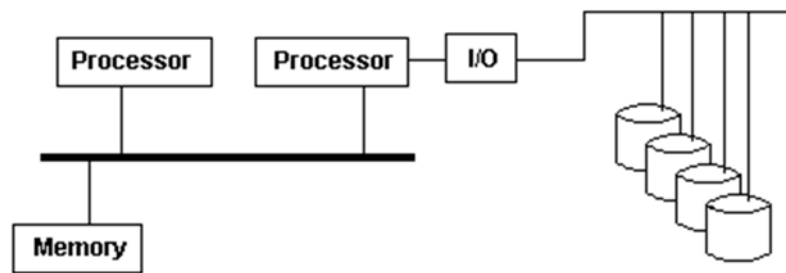
Jika fungsi-fungsi dapat disebarkan diantara banyak processor, maka kegagalan pada 1 processor tidak akan menyebabkan berhentinya system, hanya mungkin menjadi lambat.

Jika memiliki 10 processor, dan 1 fail (gagal), maka setiap dari 9 processor yang tersisa dapat mengambil alih pekerjaan processor fail tersebut.

## Jenis-jenis Multiprocessor

### 1. Asymmetric multiprocessing (AMP)

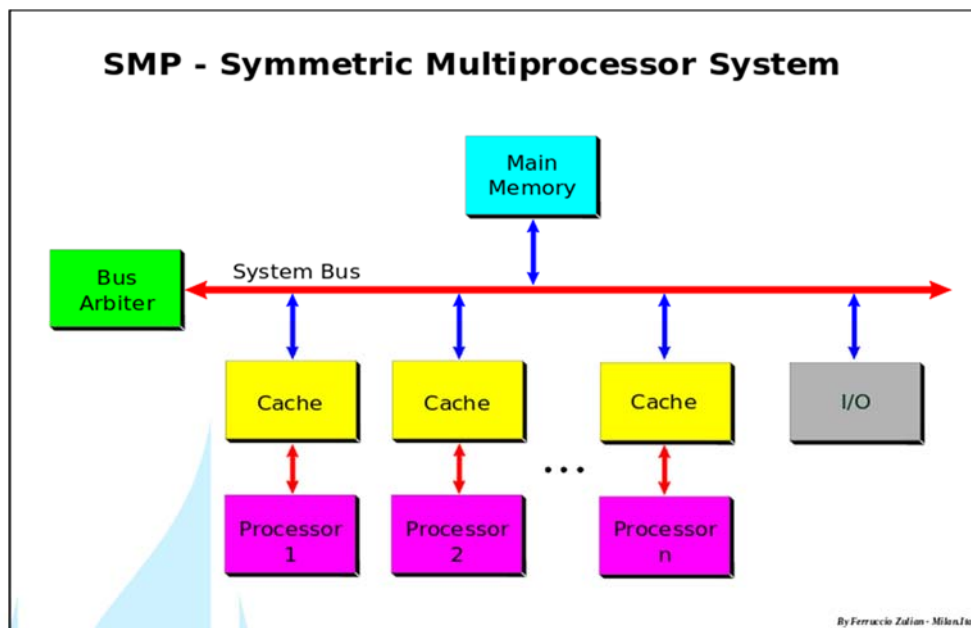
Setiap processor diberikan sebuah tugas khusus. System ini dikenal dengan nama boss-worker relationship. Hal ini disebabkan, ada sebuah boss processor yang bertugas melakukan control system, dan processor yang lain menunggu instruksi dari boss processor atau sudah diberikan tugas sejak awal.



Gambar 2.1 AMP

### 2. Symmetric multiprocessing (SMP)

System yang paling umum menggunakan symmetric multiprocessing (SMP), dimana setiap processor melaksanakan semua tugas dalam OS. Disini tidak ada boss-worker relationship antar processor, semuanya memiliki level yang sama. Gambar 2.2 memperlihatkan arsitektur SMP.



Gambar 2.2 Arsitektur SMP

### Ciri-ciri Symmetric multiprocessing (SMP)

1. Terdapat 2/lebih processor yang memiliki kemampuan yang sama
2. Processor-processor berbagi main memory yang sama, fasilitas I/O dan dihubungkan dengan sebuah bus.
3. Semua processor berbagi akses ke I/O devices, baik melalui kanal yang sama atau melalui kanal yang berbeda, tetapi menyediakan jalur ke perangkat yang sama.
4. Semua processor dapat melaksanakan fungsi yang sama (disebut symmetric ).
5. System dikontrol oleh sebuah integrated OS yang menyediakan interaksi antara processor dan program-program.

### UMA dan NUMA

---

Multiprocessing menambahkan CPU untuk meningkatkan kemampuan komputasi. Jika CPU memiliki sebuah memori controller yang terintegrasi, kemudian menambahkan CPU yang juga dapat meningkatkan jumlah memory yang dapat dialamati pada system.

Multiprocessing dapat menyebabkan sebuah system untuk bertukar akses memori dengan 2 model :

- Uniform memory access (UMA)  
Merupakan sebuah situasi dimana akses ke setiap RAM dari setiap CPU memerlukan sejumlah waktu yang sama.
- Non Uniform memory access (NUMA)  
Dengan NUMA, beberapa bagian memory, bias membutuhkan waktu yang lebih lama untuk diakses dibandingkan yang lain.

### Multicore

---

Trend desain CPU saat ini adalah multiple computing core pada sebuah chip tunggal. System multiprocessor seperti ini disebut dengan multicore.

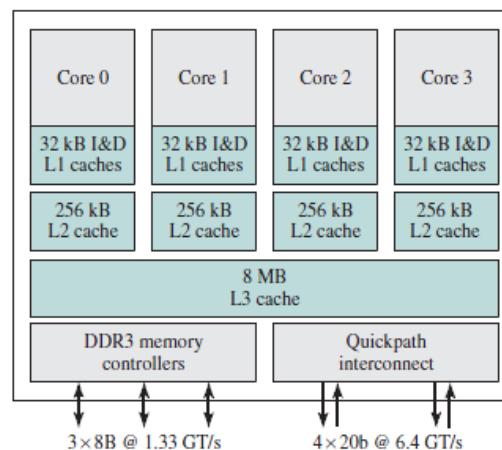
Multicore merupakan kombinasi dari 2/lebih processor (core) pada sebuah chip tunggal. Setiap core memiliki semua komponen independen processor, seperti register, ALU, Pipeline, Control Unit, dll. Gambar 2.3 merupakan contoh multicore i7.



### Keuntungan multicore

- Lebih efisien dibandingkan dengan multiple chip dengan single core, karena pada 1 chip, komunikasi bisa lebih cepat dibandingkan antar chip.
- Satu chip dengan multiple core menggunakan sedikit power dibandingkan dengan multiple single core-chip.

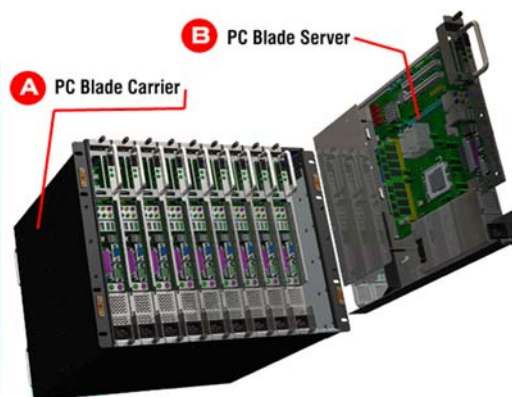
**Ctt. Tidak semua multicore systems merupakan multiprocessor systems, dan tidak semua system multiprocessor adalah multicore.**



Gambar 2.3 Blok Diagram Intel Core i7

### Blade Server

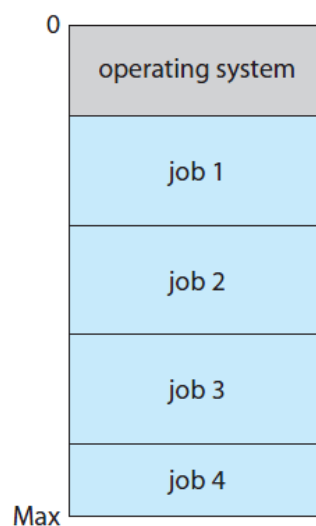
Blade server dikembangkan dengan banyak processor board, I/O board dan networking board ditempatkan dalam chassis (wadah) yang sama. Perbedaan antara blade server dengan system multiprocessor tradisional adalah setiap blade processor board, melakukan boot secara independen dan menjalankan OS nya masing-masing



# Struktur OS

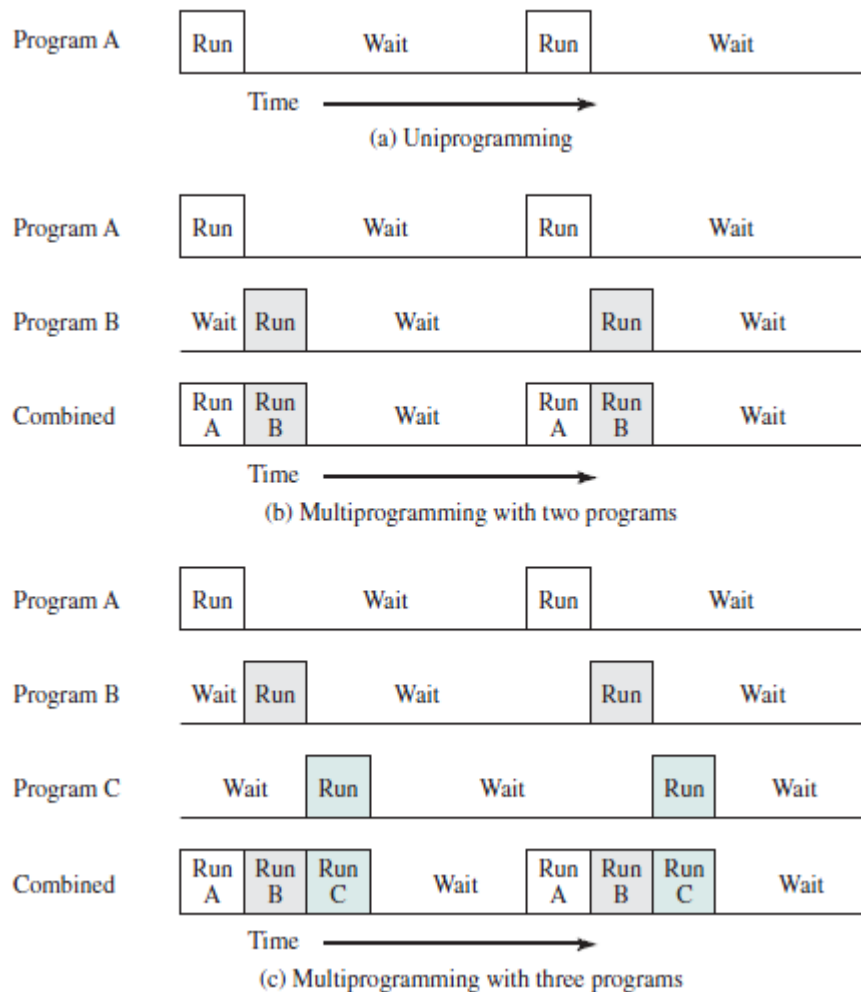
Salah satu aspek terpenting dari OS adalah kemampuannya dalam multiprogram. Sebuah program tunggal, tidak bias membuat CPU atau I/O devices menjadi sibuk sepanjang waktu. User tunggal biasanya memiliki banyak program yang berjalan. Multiprogramming dapat meningkatkan utilisasi CPU dengan mengelola job-job (kode dan data) sehingga CPU selalu memiliki job kode atau data untuk dieksekusi,

OS menyimpan beberapa job dalam memori secara bersamaan (Gambar 2.4). karena, main memory terlalu kecil untuk mengakomodasi semua job, maka job-job disimpan dulu pada disk dalam sebuah **job pool**. Pool ini berisi semua proses yang menunggu pada disk untuk dialokasikan ke main memory.



Gambar 2.4 Layout memori pada multiprogramming

## Contoh Multiprogramming



Gambar 2.6 Contoh Multiprogramming

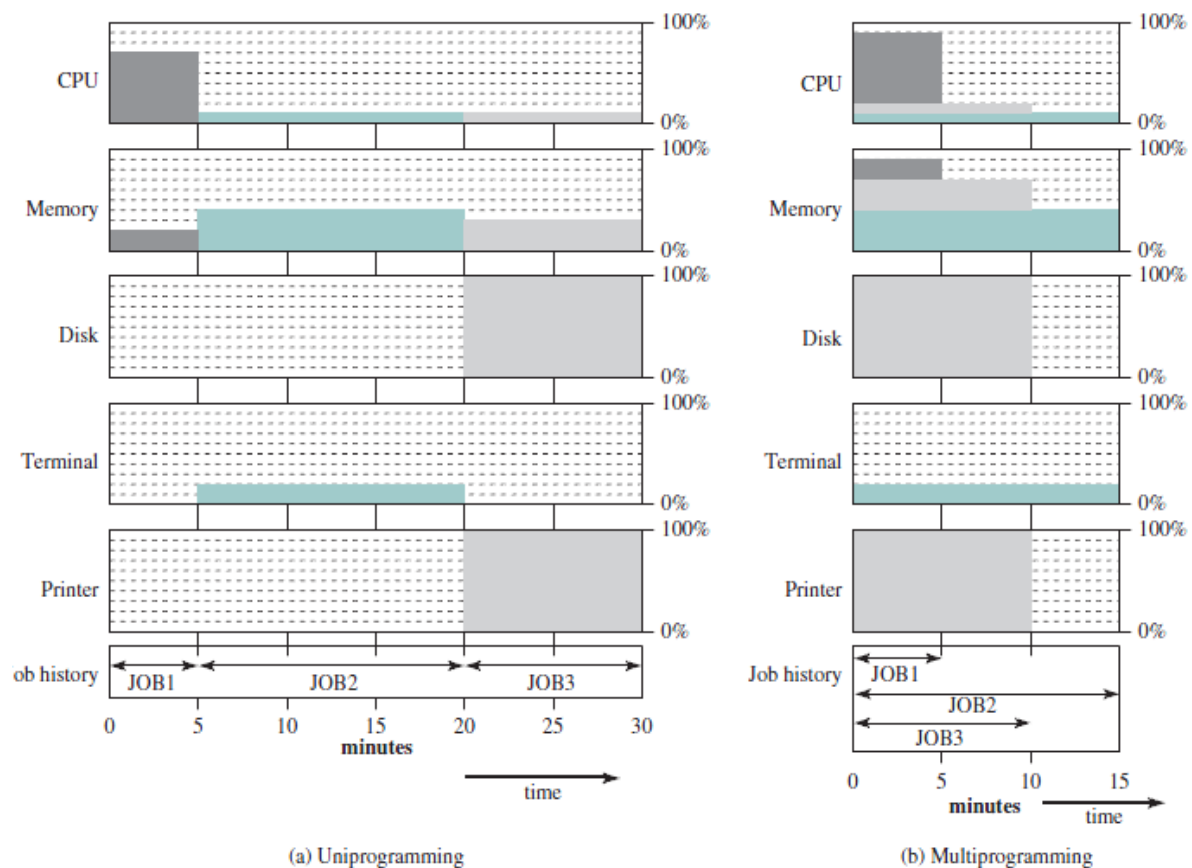
## Ilustrasi keuntungan Multiprogramming

Misalkan sebuah computer yang memiliki 250 Mbytes memory yang tersisa (tidak digunakan oleh OS), sebuah disk, sebuah terminal, dan sebuah printer. Ada 3 program JOB1, JOB2, dan JOB3 datang, untuk dieksekusi pada waktu yang sama. Tabel 2.1 adalah atribut untuk setiap job tersebut.

Table 2.1 Sample Program Execution Attributes

	JOB1	JOB2	JOB3
Type of job	Heavy compute	Heavy I/O	Heavy I/O
Duration	5 min	15 min	10 min
Memory required	50 M	100 M	75 M
Need disk?	No	No	Yes
Need terminal?	No	Yes	No
Need printer?	No	No	Yes

Utilisasi Histogram penggunaan resource antara uniprogramming dengan multiprogramming, dapat dilihat pada Gambar 2.7.



Gambar 2.7 Utilisasi resource

**Table 2.2** Effects of Multiprogramming on Resource Utilization

	Uniprogramming	Multiprogramming
Processor use	20%	40%
Memory use	33%	67%
Disk use	33%	67%
Printer use	33%	67%
Elapsed time	30 min	15 min
Throughput	6 jobs/hr	12 jobs/hr
Mean response time	18 min	10 min

# Daftar Pustaka

1. Stalling.W, *Operating System Internals and Design Principle Seventh Edition*, Prentice hall, 2012
2. Silberschatz. A, Galvin. P.B, Gagne. Greg, *Operating System Concepts Ninth perEdition*, Jhon, Wiley & Sons, 2013



## MODUL PERKULIAHAN

# Sistem Operasi

## Struktur Sistem Operasi

Fakultas  
Ilmu Komputer

Program Studi  
Teknik Informatika

Tatap Muka

**03**

Kode MK  
87030

Disusun Oleh  
Tim Dosen

### Abstract

Modul ini membahas tentang layanan system operasi, system call

### Kompetensi

Diharapkan mahasiswa dapat memahami struktur dari sebuah system operasi

# Layanan-layanan Sistem Operasi

## Tujuan Sistem Operasi

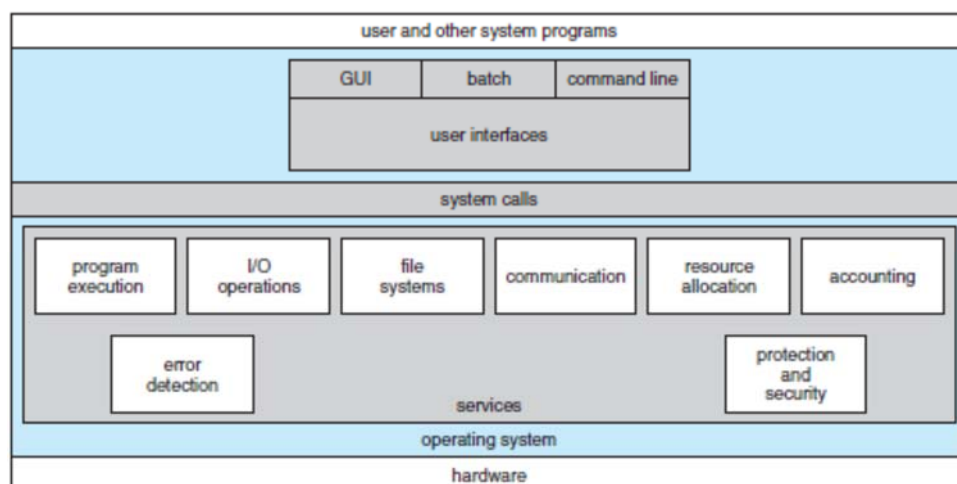
Sistem Operasi merupakan sebuah program yang mengontrol eksekusi dari program-program aplikasi dan bertindak sebagai penghubung antara aplikasi dan komputer *hardware*.

Sistem operasi memiliki 3 tujuan yaitu:

1. Kenyamanan  
OS membuat komputer lebih nyaman digunakan
2. Efisiensi  
OS mengizinkan sumber daya sistem komputer untuk digunakan lebih efisien
3. Kemampuan untuk berkembang  
OS harus dibangun dengan kondisi untuk dapat secara efektif dikembangkan, diuji dan pengenalan terhadap fungsi-fungsi dari sistem baru tanpa mengganggu layanan.

## Fungsi-fungsi Layanan System Operasi

Gambaran umum tentang fungsi-fungsi layanan dari system operasi dapat dilihat pada Gambar 3.1 dibawah ini.



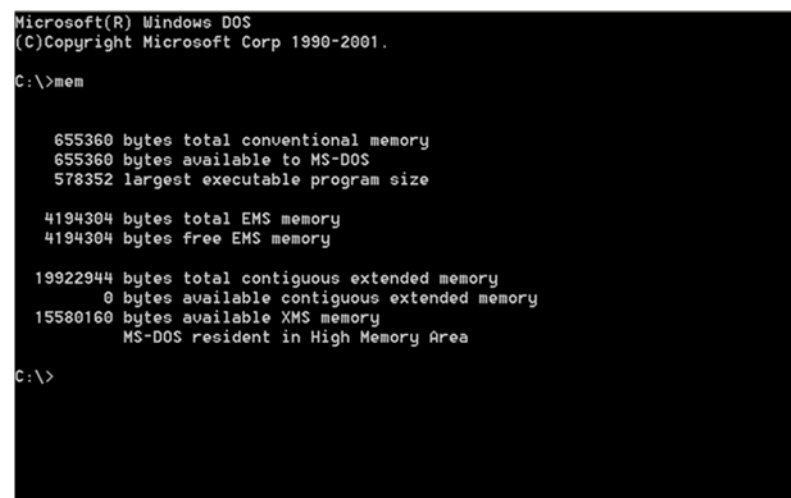
Gambar 3.1. Layanan Sistem Operasi

### 1. OS sebagai sebuah *User Interface*

Hampir semua system operasi memiliki sebuah *user interface*. *Interface* ini bermacam-macam yaitu sebagai berikut:

- Command Line Interface (CLI)

Menggunakan perintah text dan sebuah metoda untuk memasukkan perintah-perintah tersebut (misalnya keyboard untuk mengetikkan perintah dengan format tertentu). Contoh CLI adalah Ms-Dos



```
Microsoft(R) Windows DOS
(C)Copyright Microsoft Corp 1990-2001.

C:\>mem

      655360 bytes total conventional memory
      655360 bytes available to MS-DOS
      578352 largest executable program size

      4194304 bytes total EMS memory
      4194304 bytes free EMS memory

      19922944 bytes total contiguous extended memory
           0 bytes available contiguous extended memory
      15580160 bytes available XMS memory
           MS-DOS resident in High Memory Area

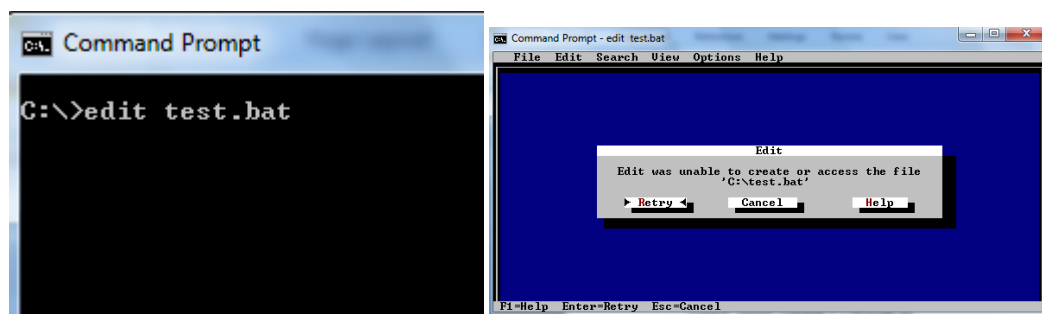
C:\>
```

Gambar 3.2. MS-DOS

- Batch Interface

Perintah-perintah dan arahan-arahan untuk mengontrol perintah-perintah tersebut dimasukkan kedalam file-file dan kemudian file-file ini dieksekusi.

Contoh seperti Gambar 3.3 dibawah ini.

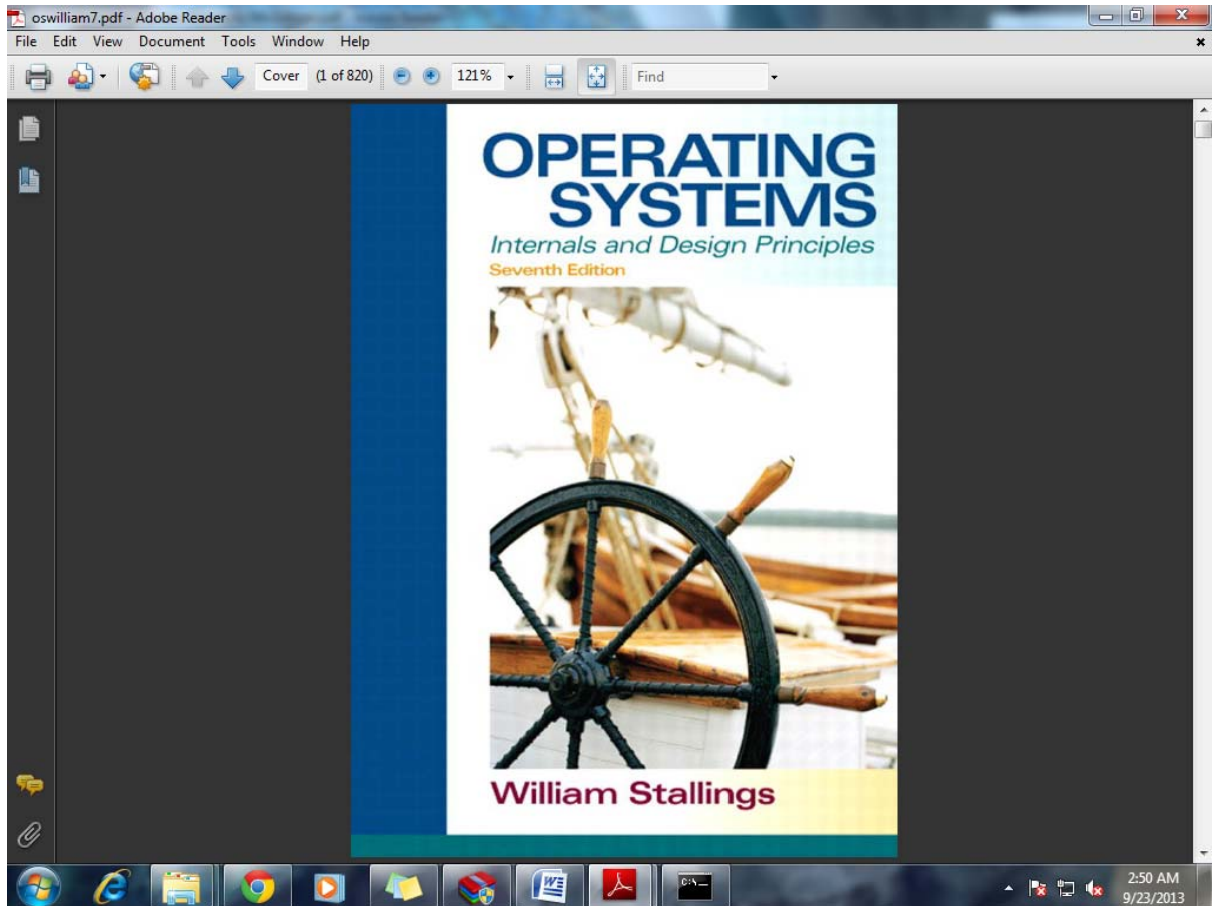


Gambar 3.3 Contoh perintah dalam Batch Interface



- Graphical User Interface (GUI)

GUI merupakan *user interface* yang paling umum digunakan. *Interface* nya merupakan sistem window dengan sebuah perangkat penunjuk arah I/O, memilih menu, dan membuat pilihan-pilihan dan sebuah keyboard untuk memasukkan text.



Gambar 3.4. GUI Windows 7

## 2. Eksekusi program

- Dibutuhkan sejumlah langkah-langkah untuk melaksanakan sebuah program
- Instruksi dan data harus di-load kedalam main memory
- Perangkat I/O dan file-file harus diinisialisasi
- Sumber daya komputer lain yang diperlukan harus disiapkan.
- Menjalankan program
- Program harus mampu untuk mengakhiri eksekusi, baik secara normal atau tidak normal (terindikasi error)

### 3. Operasi I/O

Sebuah program yang berjalan, memerlukan I/O, dimana melibatkan sebuah file atau perangkat I/O.

- Akses ke perangkat I/O

Setiap perangkat I/O membutuhkan serangkaian instruksi khusus atau sinyal control untuk beroperasi. OS menyediakan sebuah interface sama yang menyembunyikan bagian detail ini, sehingga programmer dapat mengakses perangkat-perangkat menggunakan instruksi-instruksi baca dan tulis sederhana.

- Kontrol akses ke file-file

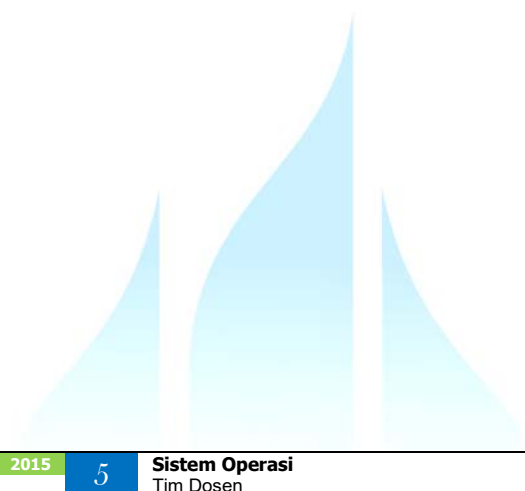
Untuk akses ke file, OS harus menggambarkan secara rinci tidak hanya sifat dasar dari perangkat I/O (harddisk) tetapi juga struktur dari data yang ada dalam file-file pada media penyimpanan. Untuk sistem dengan multiple akses, maka OS menyediakan mekanisme proteksi untuk akses control ke file-file.

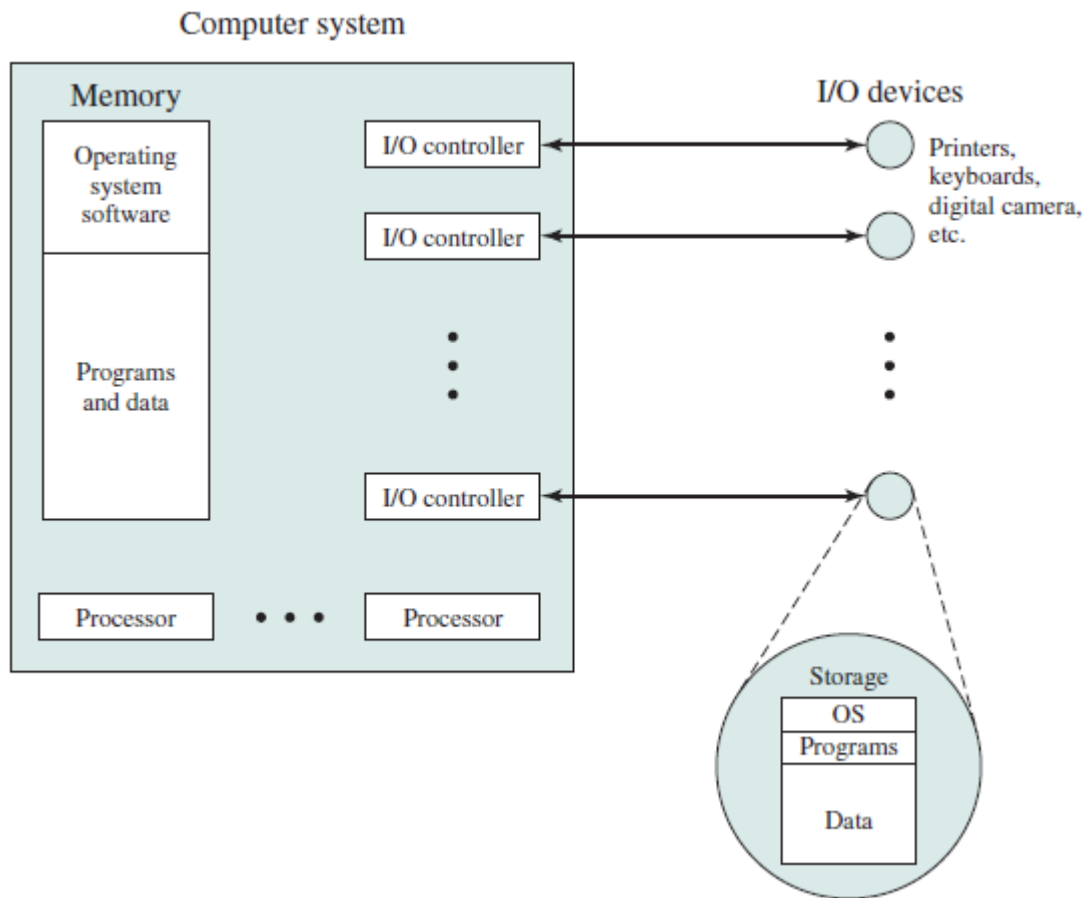
### 4. Deteksi Error

Berbagai macam error dapat terjadi pada saat sebuah sistem komputer berjalan. Error ini bisa berasal dari :

- CPU dan memory, seperti error memori atau power failure.
- Internal atau eksternal hardware, seperti pada perangkat I/O (parity error pada harddisk, koneksi yang gagal ke sebuah jaringan, kehabisan kertas pada printer
- Error pada software, seperti pembagian dengan nol, akses ke lokasi memori yang tidak diizinkan dan ketidakmampuan OS untuk melayani sebuah aplikasi.

Untuk setiap jenis error, sistem operasi harus mengambil tindakan yang tepat untuk menjamin perhitungan yang benar dan konsisten.





Gambar 3.5 OS sebagai manajer sumber daya

#### 1. Alokasi sumber daya

Sebuah komputer adalah serangkaian sumber daya yang berfungsi sebagai penggerak, media penyimpanan dan pemrosesan data serta pengontrol. OS bertanggungjawab untuk mengelola sumber daya ini.

Ketika ada multiple user dan multiple job berjalan pada saat yang sama, maka sumber daya harus dialokasikan ke masing-masingnya, maka OS yang akan mengatur nya. Contoh, bagaimana menggunakan sebuah CPU dengan baik, maka OS memiliki sebuah penjadwalan rutin yang akan menghitung kecepatan CPU, job-job yang harus dieksekusi, jumlah register yang tersedia, dan lain-lain. OS juga memiliki rutin untuk alokasi printer, USB storage drives, dan perangkat pheriperal lainnya.

## 2. Perhitungan

OS yang baik akan mengumpulkan statistik penggunaan berbagai macam sumber daya dan memonitor parameter kinerja seperti response time. Statistik penggunaan ini merupakan sebuah tool yang bermanfaat untuk peneliti yang ingin melakukan konfigurasi ulang sistem agar layanan komputasinya meningkat.

## 3. Proteksi dan keamanan

Proteksi.

Ketika beberapa proses dieksekusi secara bersamaan, maka tidak mustahil sebuah proses akan mengganggu yang lain, atau OS sendiri. OS melakukan proteksi yang menjamin bahwa semua akses ke sumber daya sistem dikontrol.

Keamanan

Pemilik informasi yang disimpan dalam sebuah sistem multiuser atau jaringan, akan mengontrol penggunaan informasi tersebut. OS menyediakan masing-masing user sebuah otentikasi.

# Operasi Sistem Operasi

Untuk menjamin eksekusi dari sistem operasi berjalan dengan benar, maka harus dibedakan antara eksekusi kode sistem operasi dan kode yang didefinisikan user. Sebagian besar sistem komputer menyediakan dukungan hardware yang membedakan berbagai macam mode eksekusi.

Ada 2 jenis mode operasi :

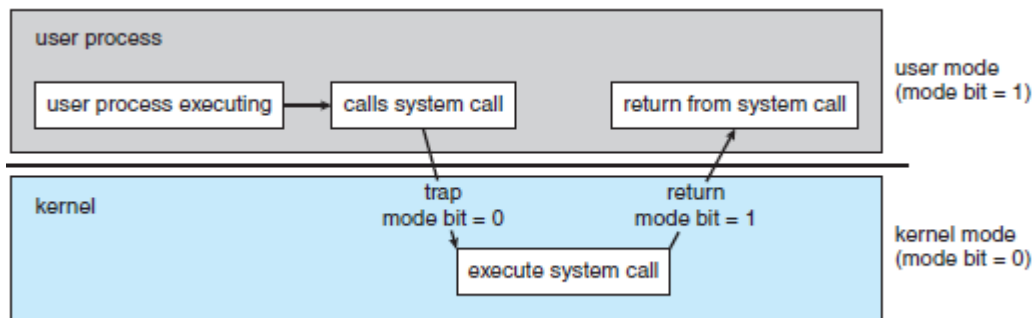
- User mode
- Kernel mode (supervisor mode, system mode atau privileged mode)

Untuk membedakan mode tersebut, maka sebuah bit (mode bit) ditambahkan ke hardware komputer untuk mengindikasikan mode tersebut:

- User → bit 1
- Kernel → bit 0

Dengan mode bit, maka dapat dibedakan apakah task yang dieksekusi adalah task OS atau aplikasi user. Pada saat sistem komputer mengeksekusi aplikasi user, maka sistem dalam mode user. Namun pada saat aplikasi user meminta layanan ke OS (melalui sebuah system

call), maka sistem harus berpindah dari mode user ke mode kernel untuk memenuhi layanan (Gambar 3.6)



Gambar 3.6 Transisi dari mode user ke mode kernel

Pada saat sistem boot, maka hardware memulai dalam kernel mode, OS kemudian di-load dan memulai aplikasi user pada user mode. Apabila sebuah trap atau interrupt terjadi, maka hardware akan berpindah dari user mode ke kernel mode (merubah state dari mode bit ke 0). Pada saat OS mendapatkan kontrol dari komputer, maka akan berada pada kernel mode. Sistem selalu memindahkan ke user mode (menset mode bit ke 1) sebelum menyerahkan kontrol ke program user.

## System Call

System call menyediakan sebuah interface kelayanan-layanan yang disediakan oleh OS. Biasanya adalah sebuah rutin yang ditulis dalam C dan C++, dan untuk tugas-tugas yang low level (seperti, tugas akses hardware secara langsung) harus ditulis menggunakan instruksi-instruksi dalam bahasa assembler.

### Ilustrasi System Call.

Membuat sebuah program sederhana untuk membaca data dari satu file dan copy data tersebut ke file lain.

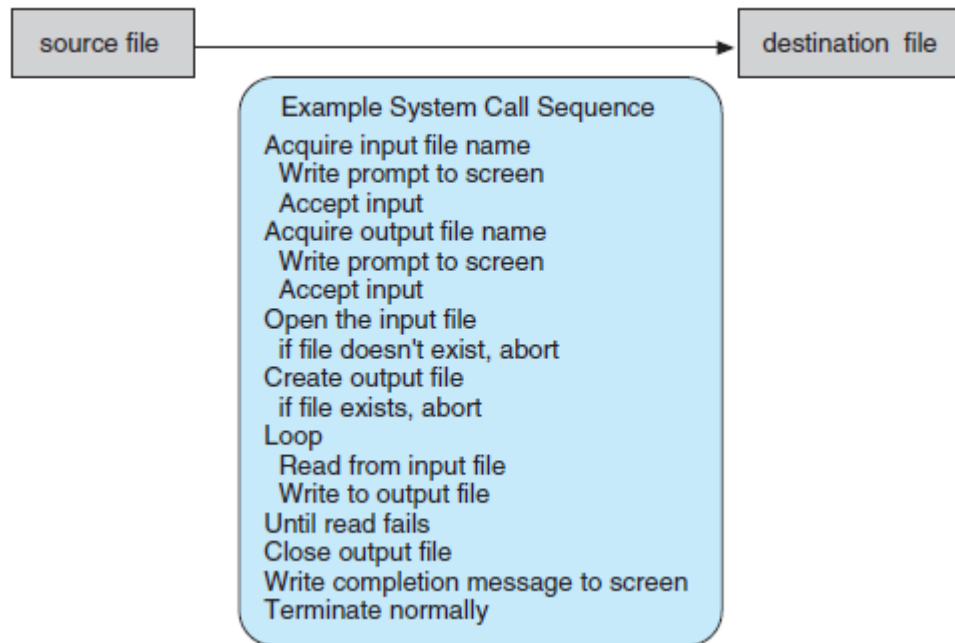
1. Input program → diperlukan nama dari 2 file yaitu input file dan output file  
Nama file ditetapkan dengan berbagai cara, tergantung dengan desain OS  
Inputan nama file bisa dilakukan dengan salah satu cara berikut:
  - Program akan meminta user.
    - Pada sistem interaktif, akan membutuhkan serangkaian system calls,
      - Menulis sebuah pesan pada layar

- Membaca dari keyboard, karakter-karakter yang didefinisikan untuk kedua file tersebut.
- Pada sistem berbasis mouse dan icon, sebuah menu dari nama-nama file biasanya ditampilkan dalam sebuah window. User dapat menggunakan mouse untuk memilih nama sumber dan sebuah window akan dibuka untuk nama tujuan yang ditetapkan.

Prosedur ini memerlukan banyak system call I/O

2. Pada saat 2 nama file telah dipilih, program harus membuka file input dan membuat file output. Masing-masing dari operasi ini membutuhkan system call yang lain. Kemungkinan error dari setiap operasi juga membutuhkan system call tambahan.
3. Pada saat program mencoba untuk membuka input file, maka ada beberapa kondisi
  - Kondisi dimana tidak menemukan nama file tersebut, atau file yang diinginkan diproteksi aksesnya, maka program harus memberikan sebuah pesan pada console (system call lain diperlukan) dan kemudian terminate secara tidak normal (system call lain diperlukan).
  - Jika input file ada, maka harus membuat sebuah output file baru. Dalam hal ini juga ada beberapa kondisi:
    - Menemukan bahwa sudah ada output file dengan nama yang sama. Maka program akan abort (system call) atau menghapus file yang ada (system call yang lain) dan membuat yang baru (system call yang lain lagi)
    - Pada sebuah sistem interaktif, meminta user (serangkaian system call yang menghasilkan sebuah pesan anjuran dan membaca respon dari terminal) apakah me-replace file yang sudah ada atau abort program.
4. Pada saat kedua file siap, maka akan masuk ke sebuah perulangan yang membaca dari input file (sebuah system call) dan menulis ke output file (system call yang lain).
5. Setelah semua file dicopy, program akan menutup kedua file (system call), menulis sebuah pesan ke console atau window (system call yang lain) dan terakhir terminate secara normal (system call yang lain).

Rangkaian system call diatas, dapat dilihat pada Gambar 3.7



Gambar 3.7 Contoh bagaimana system call digunakan

## Application Programming Interface (API)

---

Dalam sistem operasi, system akan mengeksekusi ribuan system call per detik. Sebagian besar programmer tidak pernah melihat level ini secara detail. Pengembang aplikasi mendesain program tergantung ke sebuah **Application Programming Interface (API)**. API menetapkan serangkaian fungsi-fungsi yang disediakan untuk programmer aplikasi, termasuk parameter yang diberikan untuk masing-masing fungsi, dan mengembalikan nilai-nilai yang dapat diharapkan programmer.

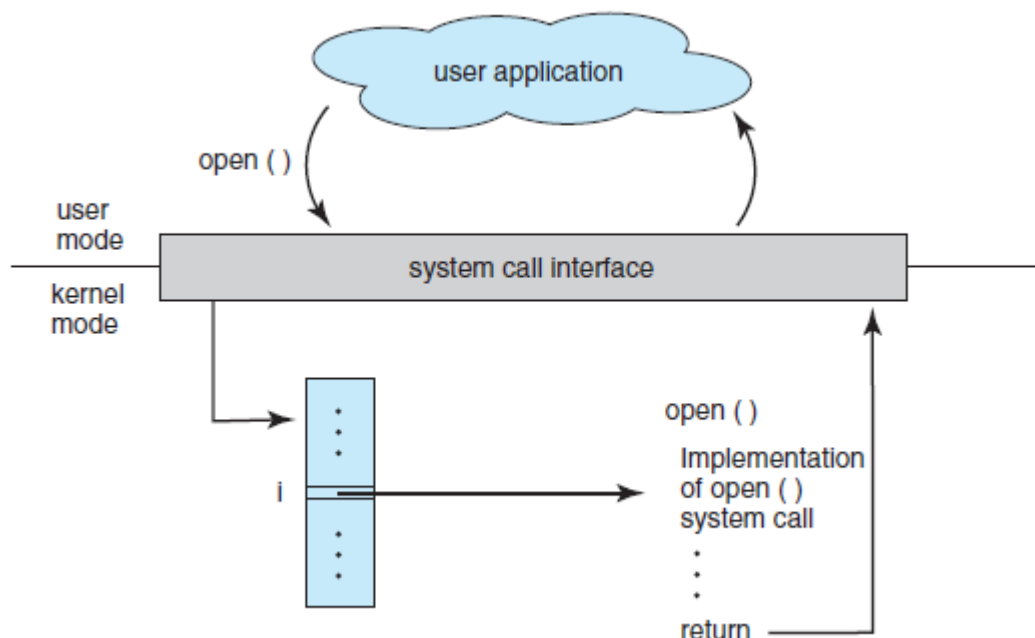
Ada 3 jenis API untuk programmer aplikasi :

- Windows API untuk system windows
- POSIX API untuk system yang berbasis POSIX (semua versi Unix, Linux dan Mac OSX)
- Java API untuk program yang berjalan pada Java Virtual Machine.

Sebagian besar bahasa pemrograman, men run-time support system (serangkaian fungsi yang dibangun menjadi library yang dilengkapi dengan sebuah compiler) menyediakan sebuah system call interface yang melayani sebagai link ke system call yang dibuat oleh sistem operasi. System call interface menangkap function call dalam API dan memanggil system call yang diperlukan dalam OS. Sebuah angka dihubungkan dengan setiap system call, dan system call interface mengelola sebuah tabel index yang berisi angka-angka ini. System call interface kemudian meminta system call yang diharapkan dalam kernel OS dan mengembalikan status dari system call dan setiap nilai kembali

Pemanggil (caller) tidak harus mengetahui semua hal tentang bagaimana system call diimplementasikan atau apa yang dilakukan selama eksekusi. Tetapi, caller hanya perlu mematuhi API dan memahami apa yang akan dilakukan OS sebagai hasil dari eksekusi system call tersebut. Sebagian besar dari detail interface OS disembunyikan dari programmer oleh API dan dikelola oleh run time support library.

Hubungan antara sebuah API, system call interface dan OS, dapat dilihat pada Gambar 3.8, yang menggambarkan bagaimana OS mengelola sebuah aplikasi user yang meminta system call `open()`.



Gambar 3.8 Penanganan aplikasi user yang meminta system call `open()`.



System call terjadi dengan cara berbeda, tergantung pada komputer yang digunakan. Ada 3 metoda umum yang digunakan untuk memberikan parameter-parameter ke OS.

1. Paling sederhana adalah memberikan parameter dalam register. Parameter secara umum disimpan dalam sebuah block, atau tabel, dalam memory dan alamat dari block diberikan sebagai sebuah parameter dalam sebuah register (Gambar 3.9). metoda ini digunakan pada Linux dan Solaris.
2. Parameter juga dapat ditempatkan atau di-Push, pada stack oleh program dan di-Pop keluar dari stack oleh OS.
3. Beberapa OS, memilih block atau metoda stack karena pendekatan-pendekatan ini tidak membatasi jumlah atau panjang parameter yang akan di berikan.

## Jenis-jenis System Call

---

System call dapat dikelompokkan menjadi 6 kategori utama:

1. Process control
2. File Manipulation
3. Device Manipulation
4. Information Maintenance
5. Communications
6. Protection

## Daftar Pustaka

1. Stalling.W, *Operating System Internals and Design Principle Seventh Edition*, Prentice hall, 2012
2. Silberschatz. A, Galvin. P.B, Gagne. Greg, *Operating System Concepts Ninth perEdition*, Jhon, Wiley & Sons, 2013



## MODUL PERKULIAHAN

# Sistem Operasi

## Proses

Fakultas

Ilmu Komputer

Program Studi

Teknik Informatika

Tatap Muka

**04**

Kode MK

87030

Disusun Oleh

Tim Dosen

### Abstract

Modul ini membahas tentang konsep dari sebuah proses

### Kompetensi

Diharapkan mahasiswa dapat memahami konsep dari sebuah proses.

# Pendahuluan

## Program vs Proses

---

Program bukan proses, dimana program merupakan serangkaian instruksi yang belum dieksekusi oleh processor dan merupakan komponen pasif.

Berikut adalah beberapa definisi dari proses.

- ✓ Sebuah program yang dieksekusi
- ✓ Sebuah program yang berjalan pada sebuah komputer
- ✓ Sebuah entitas yang diberikan ke dan dieksekusi pada processor
- ✓ Sebuah aktifitas unit yang ditandai dengan eksekusi serangkaian instruksi, state saat ini (current state) dan terhubung dengan serangkaian sumber daya sistem

Program menjadi proses apabila:

- ✓ .exe dijalankan (dieksekusi)

## Elemen-elemen proses

---

Ada 2 elemen penting dari sebuah proses yaitu:

- ✓ Kode program (bisa di share dengan proses lain yang mengeksekusi program yang sama)
- ✓ Serangkaian data yang dihubungkan dengan kode program tersebut

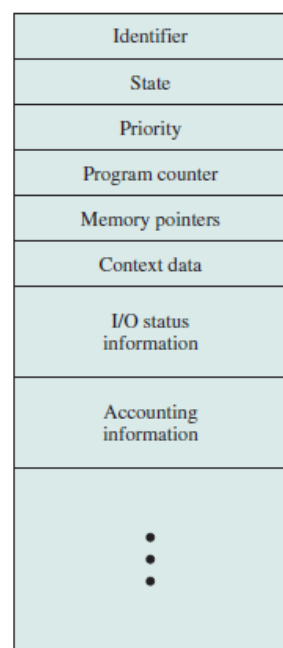
Pada saat processor mulai mengeksekusi kode program, maka program berubah menjadi proses, dimana proses akan dikenali secara dengan sejumlah elemen- elemen berikut:

- ✓ Identifier → identitas unik dari proses, untuk membedakan antara 1 proses dengan proses lain.
- ✓ State → Selama proses dieksekusi, maka akan terjadi perubahan state pada proses tersebut. State merupakan aktifitas yang sedang dilakukan oleh sebuah proses. Contoh, jika proses sedang dieksekusi, maka proses berada pada running state.
- ✓ Priority → Tingkat prioritas suatu proses terhadap proses lainnya.
- ✓ Program counter → alamat dari instruksi berikutnya dalam program yang akan dieksekusi

- ✓ Memory pointers → melibatkan pointers ke kode program dan data yang terkait dengan sebuah proses, dan juga pointer ke beberapa blok memori yang di share ke proses lain.
- ✓ Context data → data yang ada dalam register processor, pada saat proses dieksekusi.
- ✓ I/O status information → melibatkan I/O requests, I/O devices (misalnya disk drives) yang diberikan ke sebuah proses, sebuah list dari file-file yang digunakan oleh proses, dan lain-lain.
- ✓ Accounting information → melibatkan jumlah waktu processor yang digunakan, batas waktu jumlah account, dll

## Proses Control Block (PCB)

Semua informasi tentang elemen-elemen proses disimpan dalam sebuah struktur data yang disebut dengan Process Control Block (PCB), seperti pada Gambar 4.1. PCB ini dibuat dan dikelola oleh sistem operasi.



Gambar 4.1. PCB sederhana

PCB ini berisi informasi yang mencukupi, sehingga memungkinkan untuk melakukan interrupt sebuah proses yang berjalan dan kemudian mengulangi eksekusi. PCB merupakan alat kunci yang mengizinkan OS untuk dapat mendukung multiple proses dan menyediakan multiprocessing.

## ILUSTRASI

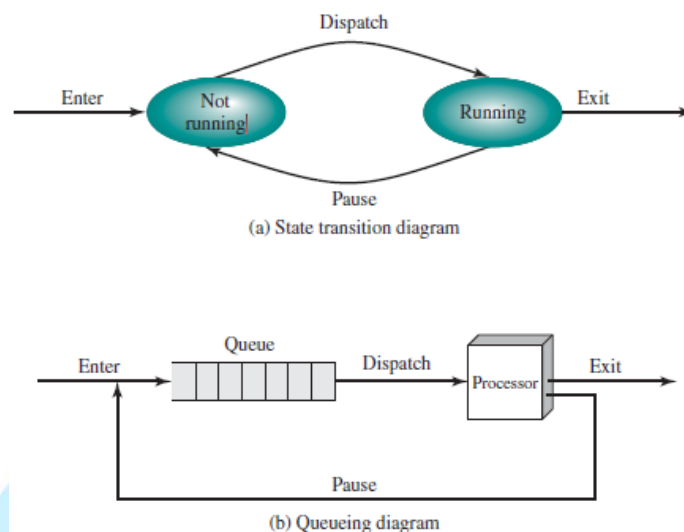
- ✓ Jika sebuah **Proses A** di-interrupt, maka nilai saat ini dari *program counter* dan register processor (Context data) dari Proses A disimpan dalam field yang terkait pada PCB, dan state Proses A dirubah ke nilai lain, misalnya di-block atau ready.
- ✓ OS, saat ini bebas untuk menempatkan proses lain (Ex. Proses B) kedalam running state.
- ✓ Program counter dan context data untuk Proses B akan diload kedalam register
- ✓ Proses B mulai dieksekusi

# State dari Proses

## 2 Model Proses State

OS bertanggung jawab untuk melakukan kontrol eksekusi proses (pola eksekusi dan alokasi sumber daya). Model yang paling sederhana adalah apakah proses sedang dieksekusi oleh processor atau tidak.

Gambar 4.2, sebuah proses bisa berada pada salah satu dari kedua state yaitu Running atau Not Running.



Gambar 4.2. 2 model proses state

- ✓ Ketika OS membuat sebuah Proses A baru, maka OS membuat sebuah PCB untuk proses A.
- ✓ State dari proses A pada Not Running State.
- ✓ Proses A dan proses-proses lain yang tidak berjalan (Not Running) disimpan pada sebuah antrian, menunggu kesempatan untuk dieksekusi. Antrian ini adalah antrian tunggal dimana setiap proses yang masuk akan ditunjuk ke sebuah PCB dari proses terkait.
- ✓ Proses yang sedang berjalan (Ex. Proses B), akan diinterrupt dan dispatcher OS akan memilih beberapa proses lain untuk dieksekusi.
- ✓ Proses B akan berpindah dari running state ke Not Running State dan salah satu proses (Misalnya Proses A terpilih) berpindah ke Running State.

## Creation and Termination Process

---

- ✓ Process Creation (Penciptaan proses)

Ketika sebuah proses baru ditambahkan, OS akan membangun sebuah struktur data yang digunakan untuk mengelola proses dan mengalokasikan ruang alamat memory untuk proses tersebut. Struktur data ini disebut dengan PCB. Hal ini disebut dengan penciptaan proses.

Ada 4 alasan penciptaan proses.

1. Sistem batch  
Sebuah proses diciptakan sebagai respon terhadap sebuah job yang diberikan.
2. Interaktif Log On  
Sebuah proses diciptakan pada saat user mencoba melakukan log-on ke sistem.
3. Dibuat oleh OS untuk menyediakan layanan  
OS bertanggung jawab untuk menciptakan sebuah proses baru yang terkait dengan sebuah aplikasi. Contoh, jika pengguna meminta sebuah file untuk di-print, maka OS membuat sebuah proses yang akan mengelola printer.
4. Berasal dari sebuah proses yang ada  
Bertujuan untuk modularity atau paralell. Akan bermanfaat untuk mengizinkan sebuah proses menciptakan proses lainnya. Sebagai contoh, sebuah proses pada server (ex, print server, file server) bisa menghasilkan proses baru untuk setiap request yang ditangani.

Pada saat OS membuat sebuah proses yang berasal dari permintaan proses lain, maka tindakan ini disebut dengan process spawning (melahirkan proses). Dimana proses yang lebih awal disebut parent proses dan proses yang dilahirkan disebut child proses.

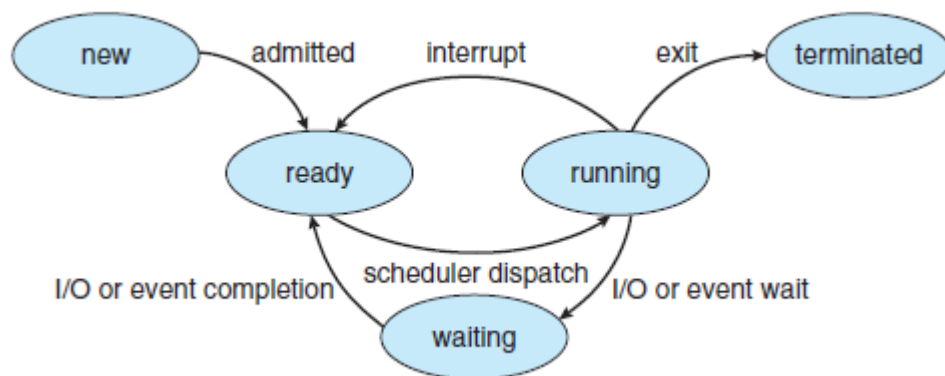
## ✓ PROCESS TERMINATION

Alasan-alasan untuk terminasi proses adalah:

Normal completion	The process executes an OS service call to indicate that it has completed running.
Time limit exceeded	The process has run longer than the specified total time limit. There are a number of possibilities for the type of time that is measured. These include total elapsed time ("wall clock time"), amount of time spent executing, and, in the case of an interactive process, the amount of time since the user last provided any input.
Memory unavailable	The process requires more memory than the system can provide.
Bounds violation	The process tries to access a memory location that it is not allowed to access.
Protection error	The process attempts to use a resource such as a file that it is not allowed to use, or it tries to use it in an improper fashion, such as writing to a read-only file.
Arithmetic error	The process tries a prohibited computation, such as division by zero, or tries to store numbers larger than the hardware can accommodate.
Time overrun	The process has waited longer than a specified maximum for a certain event to occur.
I/O failure	An error occurs during input or output, such as inability to find a file, failure to read or write after a specified maximum number of tries (when, for example, a defective area is encountered on a tape), or invalid operation (such as reading from the line printer).
Invalid instruction	The process attempts to execute a nonexistent instruction (often a result of branching into a data area and attempting to execute the data).
Privileged instruction	The process attempts to use an instruction reserved for the operating system.
Data misuse	A piece of data is of the wrong type or is not initialized.
Operator or OS intervention	For some reason, the operator or the operating system has terminated the process (e.g., if a deadlock exists).
Parent termination	When a parent terminates, the operating system may automatically terminate all of the offspring of that parent.
Parent request	A parent process typically has the authority to terminate any of its offspring.

## 5 Model State Proses

---



Gambar 4.3 State Proses

### Keterangan

- ✓ **Running** → Proses sedang dieksekusi
- ✓ **Ready** → Proses yang bersiap-siap untuk dieksekusi apabila diberikan kesempatan untuk diberikan ke CPU.
- ✓ **Blocked/Waiting** → Sebuah proses yang tidak dapat dieksekusi akan menunggu sampai beberapa event terjadi, seperti menyelesaikan operasi I/O.
- ✓ **New** → Sebuah proses yang diciptakan tetapi belum diserahkan ke pool proses yang dapat dieksekusi oleh OS. Biasanya sebuah proses baru belum diload kedalam main memory, meskipun PCB sudah dibuat
- ✓ **Terminated** → Sebuah proses yang sudah selesai

### Transisi state

---

- ✓ **Null : New**  
Sebuah proses dibuat untuk eksekusi sebuah program. Event ini terjadi untuk beberapa alasan yang sudah dijelaskan di bagian penciptaan proses diatas.
- ✓ **New : Ready**  
OS akan memindahkan sebuah proses dari new state ke ready state pada saat OS siap untuk menambah proses.
- ✓ **Ready : Running**  
Pada saat sebuah proses terpilih untuk dieksekusi, OS memilih salah satu proses yang ada pada ready state. Hal ini dilakukan oleh job scheduler atau dispatcher.



✓ Running : Terminated

Proses yang sedang berjalan, diterminasi oleh proses jika proses terindikasi sudah selesai atau jika proses tersebut dihentikan secara paksa (Abort).

✓ Running : Ready

Transisi ini terjadi biasanya pada saat proses yang berjalan sudah mencapai batasan waktu yang diizinkan untuk dieksekusi. OS memberikan proses-proses sebuah level prioritas yang berbeda-beda antara satu proses dengan proses lainnya. Misalnya proses A yang sedang diproses memiliki sebuah level prioritas yang lebih rendah dibandingkan dengan proses B datang (interrupt) dengan level prioritas yang lebih tinggi, sehingga proses A akan diblok. Hal ini disebut dengan preempted.

✓ Running : Waiting

Proses ditempatkan pada waiting state jika proses tersebut meminta sesuatu dan harus menunggu. Sebuah request ke OS biasanya adalah sebuah sistem service call, yaitu sebuah panggilan dari program yang berjalan untuk sebuah prosedur yang merupakan bagian dari kode OS.

Sebagai contoh, sebuah proses meminta layanan OS, dan OS belum siap melaksanakannya segera

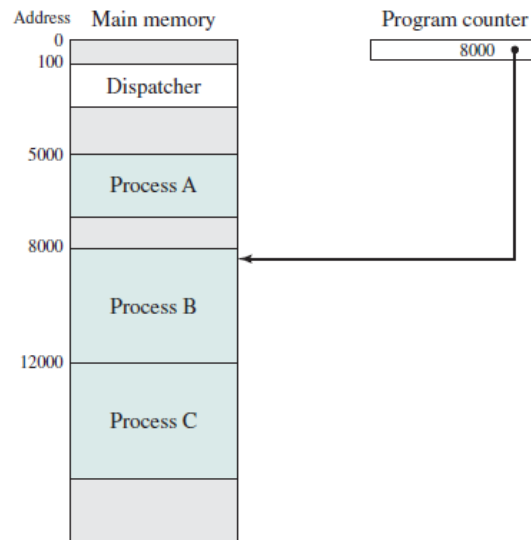
- permintaan sumber daya, file atau bagian memori yang dishare, belum tersedia
- proses yang memulai sebuah aksi, seperti operasi I/O, maka proses tersebut harus menunggu aksi selesai, sebelum dapat melanjutkan eksekusi
- sebuah proses berkomunikasi dengan proses lain, maka sebuah proses akan diblock dan menunggu proses lain untuk menyediakan data atau menunggu sebuah pesan dari proses lain

✓ Waiting : Ready

Sebuah proses dalam waiting state akan dipindahkan ke ready state jika event yang ditunggu sudah terjadi.

## CONTOH

Contoh, Gambar 4.4 memperlihatkan sebuah memory layout dari 3 proses. Ada 3 proses yang di-load ke dalam memory. Ada sebuah program dispatcher yang berfungsi melakukan switch processor dari satu proses ke proses lain.



Gambar 4.4 menunjukkan traces dari setiap proses selama bagian awal eksekusi.

5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

(a) Trace of process A      (b) Trace of process B      (c) Trace of process C

5000 = Starting address of program of process A  
8000 = Starting address of program of process B  
12000 = Starting address of program of process C

Gambar 4.5 Trace dari proses

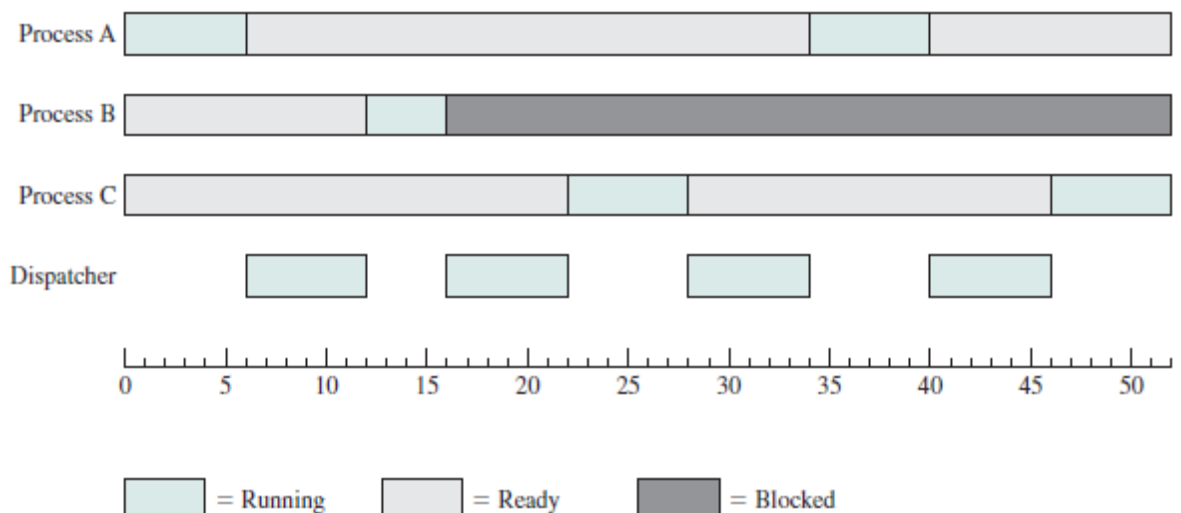
- ✓ 12 instruksi pertama dieksekusi pada proses A dan C
- ✓ Proses B meng-eksekusi 4 instruksi (asumsi 4 instruksi ini meminta operasi I/O dimana proses harus menunggu).

1	5000		27	12004
2	5001		28	12005
3	5002			-----Time-out
4	5003		29	100
5	5004		30	101
6	5005		31	102
	-----Time-out		32	103
7	100		33	104
8	101		34	105
9	102		35	5006
10	103		36	5007
11	104		37	5008
12	105		38	5009
13	8000		39	5010
14	8001		40	5011
15	8002			-----Time-out
16	8003		41	100
	-----I/O request		42	101
17	100		43	102
18	101		44	103
19	102		45	104
20	103		46	105
21	104		47	12006
22	105		48	12007
23	12000		49	12008
24	12001		50	12009
25	12002		51	12010
26	12003		52	12011
				-----Time-out

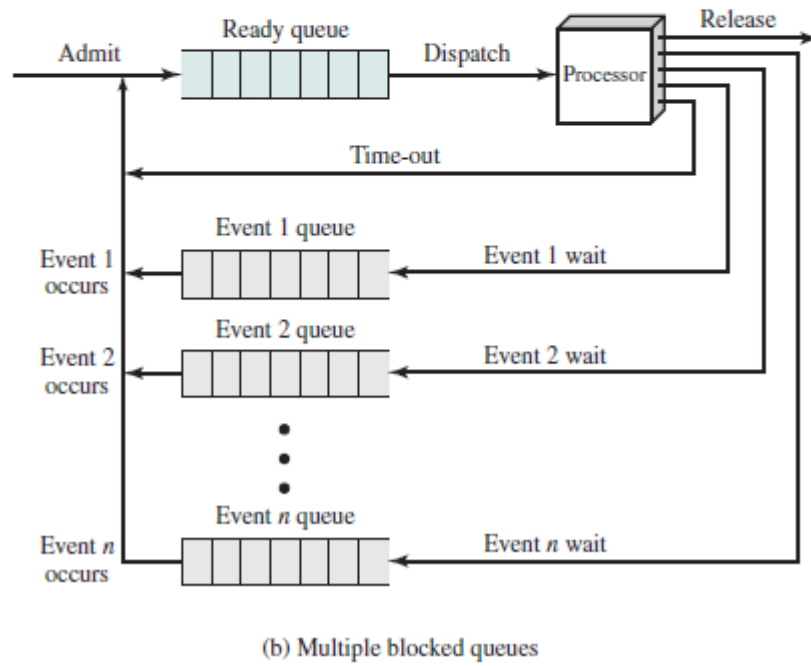
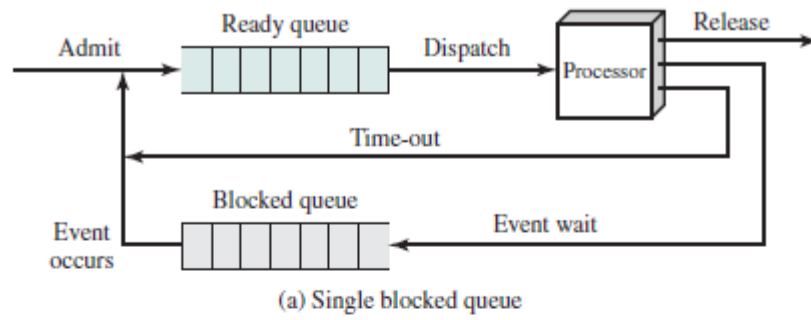
100 = Starting address of dispatcher program

Shaded areas indicate execution of dispatcher process;  
first and third columns count instruction cycles;  
second and fourth columns show address of instruction being executed

Gambar 4.6 Trace proses



Gambar 4.7. proses state dari trace proses pada Gambar 4.6



Gambar 4.8. Model antrian gambar 4.6

# Daftar Pustaka

1. Stalling.W, *Operating System Internals and Design Principle Seventh Edition*, Prentice hall, 2012
2. Silberschatz. A, Galvin. P.B, Gagne. Greg, *Operating System Concepts Ninth perEdition*, Jhon, Wiley & Sons, 2013



## MODUL PERKULIAHAN

# Sistem Operasi

## Proses Bagian II

Fakultas  
Ilmu Komputer

Program Studi  
Teknik Informatika

Tatap Muka

**05**

Kode MK  
87030

Disusun Oleh  
Tim Dosen

### Abstract

Modul ini membahas tentang konsep dari sebuah proses

### Kompetensi

Diharapkan mahasiswa dapat memahami konsep dari sebuah proses.

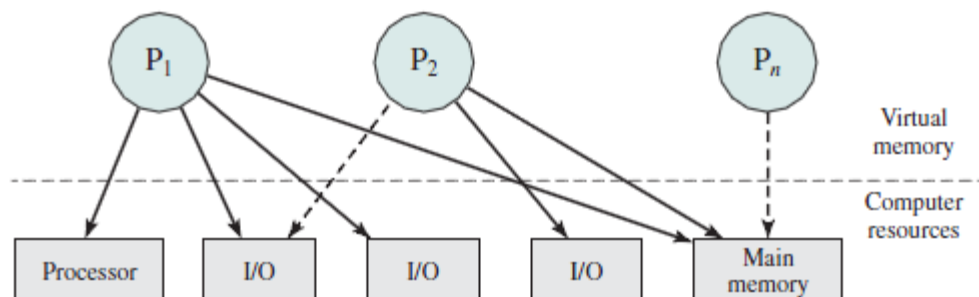
# Deskripsi Proses

OS bertugas mengontrol event yang terjadi pada system computer. Dimana OS menjadwalkan dan memerintahkan proses untuk dieksekusi oleh processor, mengalokasikan sumber daya untuk proses dan merespon permintaan pengguna proses untuk layanan-layanan dasar. Pada dasarnya OS merupakan sebuah entitas yang mengelola penggunaan dari sumber daya system oleh banyak proses.

Contoh pada Gambar 5.1, pada sebuah multiprogramming, ada sejumlah proses ( $P_1, \dots, P_n$ ) yang sudah diciptakan dan berada dalam virtual memory. Setiap proses, selama eksekusi memerlukan akses ke suber daya system tertentu, termasuk processor, perangkat I/O, dan main memory.

Pada gambar terlihat :

- ✓ Proses  $P_1$  running, sebagian dari proses berada dalam main memori, dan memiliki control 2 perangkat I/O.
- ✓ Proses  $P_2$  berada dalam main memory tapi di blok, kerana menunggu sebuah perangkat I/O yang sudah dialokasikan untuk  $P_1$ .
- ✓ Proses  $P_n$  sudah di swap out (dikeluarkan dari memory) dan disingkirkan.



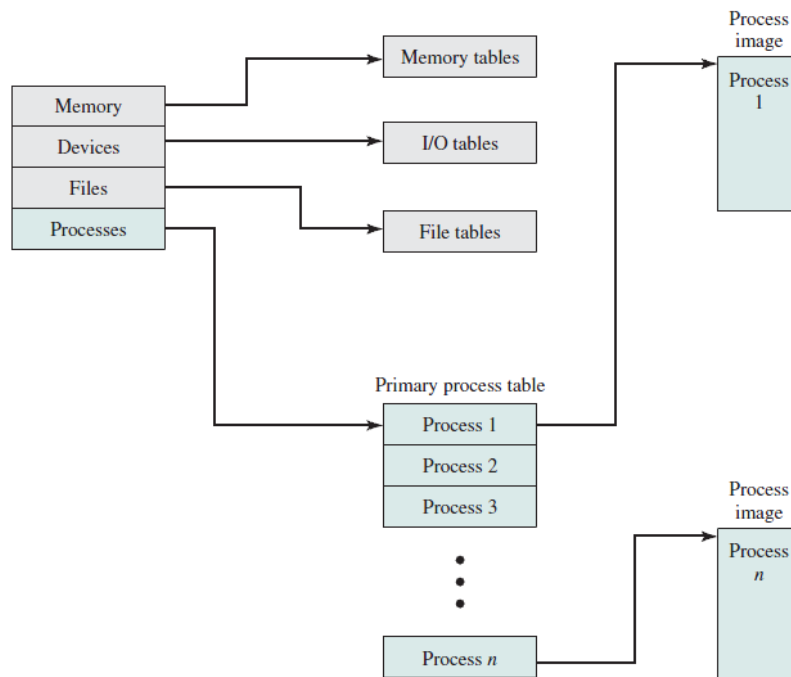
Gambar 5.1 Proses dan sumber daya

## Struktur Kontrol OS

Jika OS adalah pengelola proses dan sumber daya, maka OS harus memiliki informasi tentang status saat ini dari setiap proses dan sumber daya. Pendekatan umum untuk menyediakan informasi ini adalah **straightforward**.

Straightforward adalah sebuah metoda dimana OS membangun dan menjaga table informasi tentang setiap entitas yang dikelola.

Gambar 5.2, terlihat ada 4 jenis table berbeda, yang dikelola oleh OS yaitu memori, I/O, file dan proses.



**Gambar 5.2 Struktur umum table control OS**

#### ✓ Tabel memory

- Digunakan untuk menyimpan track dari main memory (real) dan secondary memory (virtual).
- Beberapa bagian dari main memory disediakan untuk digunakan oleh OS, dan bagian yang lain digunakan untuk proses.
- Proses-proses dikelola pada secondary memory.
- Table memory harus berisi informasi :
  - Alokasi main memory untuk proses-proses
  - Alokasi secondary memory untuk proses
  - Atribut-atirbut proteksi untuk blok main/virtual memory, seperti proses-proses mana yang boleh mengakses bagian memori yang di-share.
  - Informasi yang diperlukan untuk mengelola virtual memory.

#### ✓ Tabel I/O

Digunakan OS untuk mengelola perangkat-perangkat I/O dan kanal-kanal dari sistem komputer. Sebuah perangkat I/O bisa saja tersedia atau sedang diberikan ke sebuah proses tertentu, jika sebuah operasi I/O sedang berjalan, OS harus mengetahui status dari operasi I/O dan lokasi main memory yang sedang digunakan sebagai sumber atau tujuan dari transfer I/O.



✓ **Tabel File**

Table ini menyediakan informasi tentang file-file yang ada, lokasi file pada secondary memory, status file saat ini, dan atribut lainnya.

✓ **Tabel proses**

Table proses digunakan untuk mengelola proses.

## Process Control Structures

Ada beberapa hal yang OS harus ketahui jika ingin melakukan manajemen dan mengontrol sebuah proses, yaitu:

- OS harus mengetahui dimana proses ditempatkan
- OS harus mengetahui atribut dari proses yang dikelola (ex. Proses ID dan state proses)

### Lokasi Proses

---

Sebuah proses melibatkan sebuah program atau serangkaian program yang dieksekusi. Berhubungan dengan program ini adalah serangkaian data yang ditempatkan pada variable local dan global dan juga didefinisikan sebagai konstanta. Sebuah proses akan terdiri dari memory yang cukup untuk menampung program dan data dari proses tersebut. Eksekusi program biasanya melibatkan stack, berfungsi menyimpan track prosedur call dan pemberian parameter antar prosedur. Setiap proses harus dihubungkan dengan sejumlah attribute yang digunakan OS untuk control proses, yang disebut sebagai PCB.

Kumpulan program, data, stack, dan atribut diatas disebut sebagai process image (Table 5.1)

**Tabel 5.1 Process Image**

<b>User Data</b> The modifiable part of the user space. May include program data, a user stack area, and programs that may be modified.
<b>User Program</b> The program to be executed.
<b>Stack</b> Each process has one or more last-in-first-out (LIFO) stacks associated with it. A stack is used to store parameters and calling addresses for procedure and system calls.
<b>Process Control Block</b> Data needed by the OS to control the process (see Table 3.5).

Lokasi dari sebuah process image akan tergantung pada bentuk manajemen memori yang digunakan. Pada contoh sederhana, process image dikelola secara bersebelahan atau kontinyu, blok memory. Blok ini berada dalam memori sekunder (disk). OS dapat mengelola proses, minimal sebagian kecil dari process image tersebut harus ada pada main memory. Untuk eksekusi proses, seluruh process image harus di-load ke dalam main memory atau setidaknya pada virtual memory. Sehingga OS harus mengetahui lokasi dari setiap proses pada disk dan pada main memory.

## Atribut Proses

---

Informasi pada PCB dapat dibagi menjadi 3 kategori, yaitu

- **Process identification (PID).**  
Setiap proses diberikan sebuah pengenal berupa sebuah angka unik (bilangan integer), yang digunakan sebagai indeks untuk mengakses berbagai atribut sebuah proses.
- **Processor state information**  
Merupakan isi dari register processor. Pada saat sebuah proses berjalan, maka informasinya ada pada register. Ketika sebuah proses di-interrupt, semua informasi pada register harus disimpan, sehingga dapat di-restore apabila proses dieksekusi kembali. Jumlah dari register, tergantung kepada desain processor. Umumnya, register terdiri dari:
  - User visible registers
  - Control registers
  - Status registers
  - Stack pointers.
- **Process control information**  
Merupakan informasi tambahan yang diperlukan oleh OS untuk melakukan control dan koordinasi berbagai proses aktif.

Penjelasan lebih lengkapnya disajikan pada Tabel 5.2 yang merupakan list kategori informasi yang dibutuhkan OS untuk setiap proses.

Tabel 5.2 Kategori informasi proses

Process Identification
<p><b>Identifiers</b></p> <p>Numeric identifiers that may be stored with the process control block include</p> <ul style="list-style-type: none"> <li>• Identifier of this process</li> <li>• Identifier of the process that created this process (parent process)</li> <li>• User identifier</li> </ul>
Processor State Information
<p><b>User-Visible Registers</b></p> <p>A user-visible register is one that may be referenced by means of the machine language that the processor executes while in user mode. Typically, there are from 8 to 32 of these registers, although some RISC implementations have over 100.</p> <p><b>Control and Status Registers</b></p> <p>These are a variety of processor registers that are employed to control the operation of the processor. These include</p> <ul style="list-style-type: none"> <li>• <b>Program counter:</b> Contains the address of the next instruction to be fetched</li> <li>• <b>Condition codes:</b> Result of the most recent arithmetic or logical operation (e.g., sign, zero, carry, equal, overflow)</li> <li>• <b>Status information:</b> Includes interrupt enabled/disabled flags, execution mode</li> </ul> <p><b>Stack Pointers</b></p> <p>Each process has one or more last-in-first-out (LIFO) system stacks associated with it. A stack is used to store parameters and calling addresses for procedure and system calls. The stack pointer points to the top of the stack.</p>
Process Control Information
<p><b>Scheduling and State Information</b></p> <p>This is information that is needed by the operating system to perform its scheduling function. Typical items of information:</p> <ul style="list-style-type: none"> <li>• <b>Process state:</b> Defines the readiness of the process to be scheduled for execution (e.g., running, ready, waiting, halted).</li> <li>• <b>Priority:</b> One or more fields may be used to describe the scheduling priority of the process. In some systems, several values are required (e.g., default, current, highest-allowable).</li> <li>• <b>Scheduling-related information:</b> This will depend on the scheduling algorithm used. Examples are the amount of time that the process has been waiting and the amount of time that the process executed the last time it was running.</li> <li>• <b>Event:</b> Identity of event the process is awaiting before it can be resumed.</li> </ul> <p><b>Data Structuring</b></p> <p>A process may be linked to other process in a queue, ring, or some other structure. For example, all processes in a waiting state for a particular priority level may be linked in a queue. A process may exhibit a parent-child (creator-created) relationship with another process. The process control block may contain pointers to other processes to support these structures.</p> <p><b>Interprocess Communication</b></p> <p>Various flags, signals, and messages may be associated with communication between two independent processes. Some or all of this information may be maintained in the process control block.</p> <p><b>Process Privileges</b></p> <p>Processes are granted privileges in terms of the memory that may be accessed and the types of instructions that may be executed. In addition, privileges may apply to the use of system utilities and services.</p> <p><b>Memory Management</b></p> <p>This section may include pointers to segment and/or page tables that describe the virtual memory assigned to this process.</p> <p><b>Resource Ownership and Utilization</b></p> <p>Resources controlled by the process may be indicated, such as opened files. A history of utilization of the processor or other resources may also be included; this information may be needed by the scheduler.</p>



# Kontrol Proses

## Penciptaan Proses

---

Pada saat OS menciptakan sebuah proses baru, maka yang terjadi adalah sebagai berikut:

1. Memberikan sebuah proses identifier unik untuk setiap proses baru.

Sebuah entry baru akan ditambahkan ke tabel proses, yang berisi satu entry per proses.

2. Alokasi ruang untuk proses.

Melibatkan semua elemen dari proses image. OS harus mengetahui seberapa banyak ruang yang diperlukan untuk ruang alamat private user (program dan data) dan user stack. Nilai ini akan diberikan berdasarkan jenis dari proses, atau dapat diset berdasarkan permintaan user pada saat job diciptakan. Jika sebuah proses dilahirkan oleh proses lain, parent proses dapat memberikan nilai yang diperlukan ke OS sebagai bagian dari permintaan penciptaan proses. Jika ruang alamat tersedia, akan di-share ke proses baru, sebuah link di-set up. Ruang untuk sebuah PCB juga dialokasikan.

3. Inisialisasi PCB

Bagian identifikasi proses berisi ID dari proses + IDs terkait (parent process). Bagian processor state information akan meng-inisialisasi sebagian besar dengan nol (0), kecuali program counter (set ke program entry point) dan system stack pointer (set ke nilai batas process stack).

Processor state information, di-inisialisasi berdasarkan pada standar nilai default + attribut yang diminta untuk sebuah proses.

Sebagai contoh. Processe state akan diinisialisasi ke Ready atau Ready/suspend. Prioritas akan diset secara default ke prioritas terendah, kecuali untuk sebuah request tertentu yang dibuat untuk prioritas yang lebih tinggi.

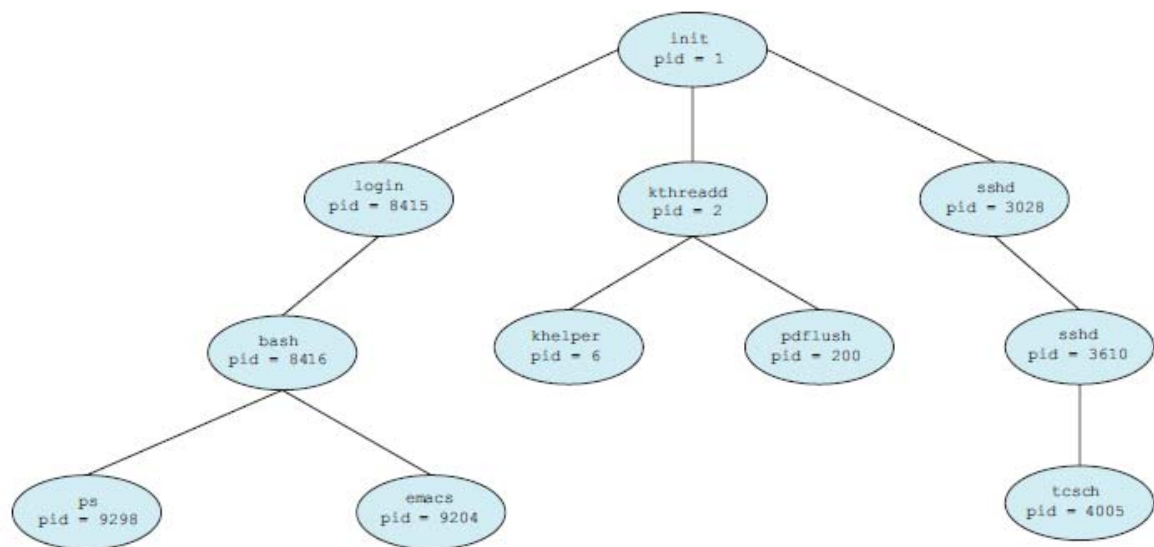
Pada awalnya proses tidak memiliki sumber daya (I/O device, file), kecuali ada permintaan khusus atau warisan dari parent proses.

4. Set linkages yang sesuai.

Jika OS mengelola setiap penjadwalan antrian sebagai linked list, maka sebuah proses baru harus ditempatkan pada ready list.

5. Create atau expand struktur data lain.

OS akan mengelola sebuah file penghitung pada setiap proses yang digunakan sebagai billing dan/atau tujuan pengujian kinerja.



**Gambar 5.3. Proses tree pada sistem linux**

Gambar 5.3 memperlihatkan sebuah pohon proses pada OS Linux, yang menunjukkan nama dari setiap proses dan PID nya. Init process (selalu memiliki PID 1), melayani sebagai root parent process untuk semua proses user. Pada saat sistem sudah booting, init process dapat juga menciptakan berbagai proses user, seperti web atau print server (ssh server).

Terlihat 2 children dari init yaitu kthreadd dan sshd. Kthreadd process bertanggung jawab untuk menciptakan proses tambahan yang melaksanakan tugas-tugas pada bagian kernel (ex. Khelper dan pdflush), sshd process bertanggung jawab untuk menangani client-client yang terhubung dengan system menggunakan ssh (secure shell). Login process bertanggung jawab untuk mengelola client yang log on dan menggugurkan bash shell, yang diberikan PID 8416. Menggunakan bash command line interface, user ini menciptakan process ps dan emacs.

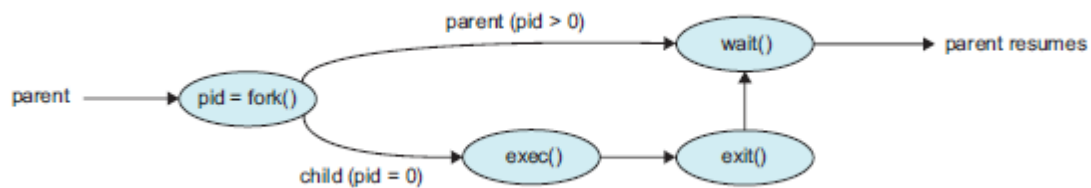
Ketika sebuah proses menciptakan child process, maka child process akan memerlukan sumber daya tertentu (CPU time, memori, file, I/O device) untuk menyelesaikan tugasnya. Child process dapat mendapatkan sumber daya secara langsung dari OS atau dibatasi sebagai bagian dari sumber daya parent process. Parent bisa membatasi sumber daya yang dimiliki diantara children proses, atau dapat berbagi sumber daya (memory atau file) diantara beberapa children nya. Pembatasan child process ke sumber daya parent mencegah setiap proses dari overload sistem karena menciptakan terlalu banyak child proses.

Pada saat sebuah proses menciptakan sebuah proses baru, 2 kemungkinan akan terjadi:

1. Parent melanjutkan eksekusi bersama dengan children nya.
2. Parent menunggu sampai beberapa atau semua children sudah selesai (terminated).

Ada 2 kemungkinan ruang alamat untuk sebuah proses baru:

1. Child process akan menduplikat parent process (memiliki program dan data yang sama dengan parent).
2. child process memiliki sebuah program baru yang di-load.



Gambar 5.4 process creation menggunakan fork() system call

## Terminasi Proses & Switching

---

SILAHKAN DIBACA BUKU TEKS PADA DAFTAR PUSTAKA, DISKUSIKAN DI FORUM.

# Daftar Pustaka

1. Stalling.W, *Operating System Internals and Design Principle Seventh Edition*, Prentice hall, 2012
2. Silberschatz. A, Galvin. P.B, Gagne. Greg, *Operating System Concepts Ninth perEdition*, Jhon, Wiley & Sons, 2013





## MODUL PERKULIAHAN

# Sistem Operasi

## THREAD

Fakultas  
Ilmu Komputer

Program Studi  
Teknik Informatika

Tatap Muka

06

Kode MK  
87030

Disusun Oleh  
Tim Dosen

### Abstract

Modul ini membahas tentang thread

### Kompetensi

Diharapkan mahasiswa dapat memahami konsep thread

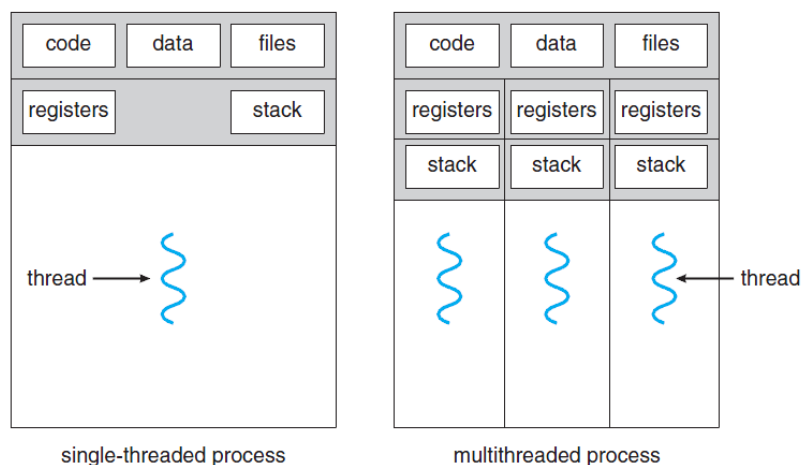


# Defenisi

Thread merupakan sebuah unit dasar dari penggunaan CPU. Sebuah thread terdiri dari thread ID, program counter, serangkain register, dan stack.

Sebuah proses tradisional (**heavyweight**) memiliki sebuah thread kontrol tunggal. Jika sebuah proses memiliki banyak thread kontrol, maka dapat melaksanakan banyak tugas pada satu waktu.

Gambar 6.1 memperlihatkan perbedaan antara proses dengan thread tunggal dan multithreaded.



**Gambar 6.1 Proses dan sumber daya**

## Contoh Multithread

### 1. Web Browser

Sebuah web browser memiliki banyak thread, misalnya 1 thread untuk menampilkan gambar, 1 thread untuk menampilkan teks, 1 thread untuk mengambil data dari jaringan.

### 2. Word Processor

Word processor bias memiliki 1 thread untuk menampilkan grafik, 1 thread untuk membaca keyboard yang ditekan user, 1 thread untuk melaksanakan pengecekan spelling dang rammer.

### 3. Web Server

Web server menerima permintaan client untuk sebuah halaman web, gambar, suara dan lain-lain. Untuk sebuah web server, bias diakses oleh ribuan bahkan jutaan client pada saat bersamaan.

Jika web server berjalan sebagai sebuah proses dengan thread tunggal, maka kemampuan untuk memberikan layanan hanya satu client per waktu.

### **Masalah:**

Client yang lain akan menunggu untuk waktu yang sangat lama agar request nya dilayani.

### **Solusi:**

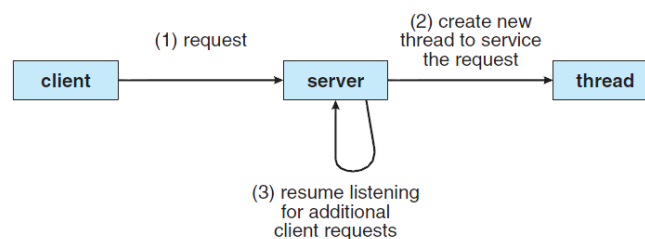
- Process creation

Server berjalan sebagai sebuah proses tunggal yang menerima request. Pada saat server menerima sebuah request, maka akan tercipta proses lain untuk menerima request tersebut. Sehingga jika ada 100 request, maka akan ada 100 proses baru yang tercipta. Hal ini akan menghabiskan waktu dan sumber daya.

**Noted. Process creation digunakan sebelum ada thread**

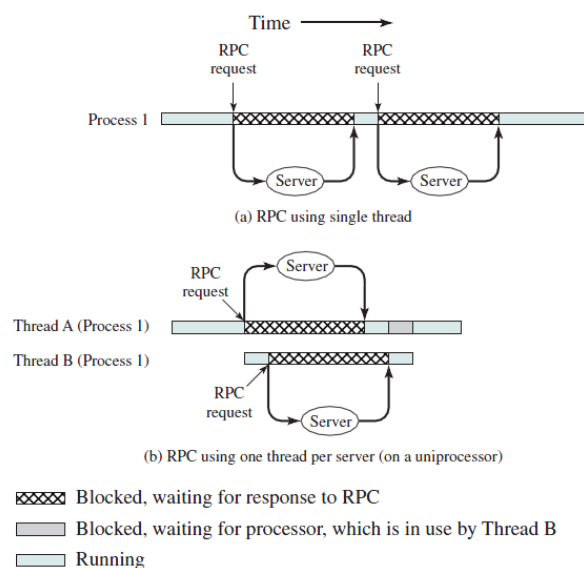
- Multithreading

Jika proses web server adalah multithread, server akan membuat sebuah thread lain yang bertugas mendengarkan permintaan client. Pada saat sebuah request dibuat, server akan membuat sebuah thread baru untuk melayani request tersebut dan server akan kembali mendengarkan request lainnya. (Gambar 6.2)



**Gambar 6.2 Arsitektur Multithread Server**

Dalam hal ini, aturan yang digunakan adalah system RPC (Remote Procedure Call)



**Gambar 6.3 RPC menggunakan thread**

## State dari thread

---

State kunci dari thread adalah **Running, Ready dan Block**.

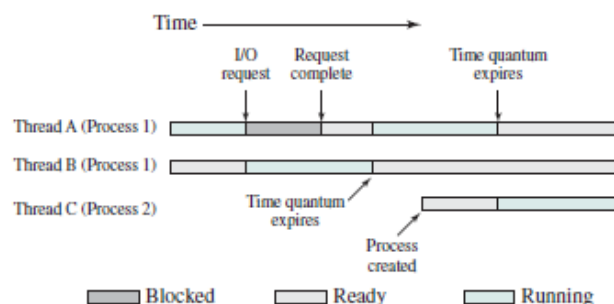
Ada 4 dasar operasi thread yang terkait dengan perubahan state pada thread:

- **Spwan**  
Pada saat sebuah proses baru di-lahirkan (spaw), sebuah thread untuk proses tersebut juga diciptakan. Sebuah thread dalam sebuah proses bisa di lahirkan oleh thread lain dalam proses yang sama, menyediakan sebuah instruksi pointer dan argument untuk thread baru. Thread baru diberikan register context dan stack space sendiri dan ditempatkan pada ready queue.
- **Block**  
Apabila sebuah thread harus menunggu sebuah event, maka thread tersebut akan diblok (menyimpan user register, program counter dan stack pointer). Processor akan mengeksekusi thread lain yang siap baik dari proses yang sama atau berbeda.
- **Unblock**  
Jika sebuah event dari thread yang diblock terjadi, maka thread tersebut dipindahkan ke ready queue.
- **Finish**  
Pada saat sebuah thread selesai, register context dan stacks nya di alokasikan ke thread yang lain.

## Multithreading pada uniprocessor

---

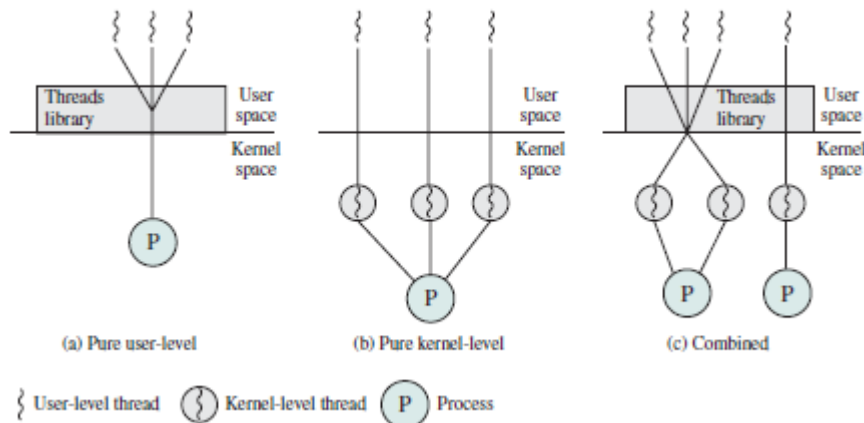
Pada Gambar 6.7, terlihat 3 threads dalam 2 proses. Eksekusi dikerjakan dari 1 thread ke thread yang lain pada saat thread berjalan saat ini di block atau pada saat time slice sudah habis.



Gambar 6.7 Multithreading pada uniprocessor

# Jenis – jenis thread

Ada 2 jenis thread yaitu user-level threads (ULTs) dan kernel-level threads (KLTs).



**Gambar 6.8 User Level dan Kernel Level Threads**

## User Level Threads (ULT)

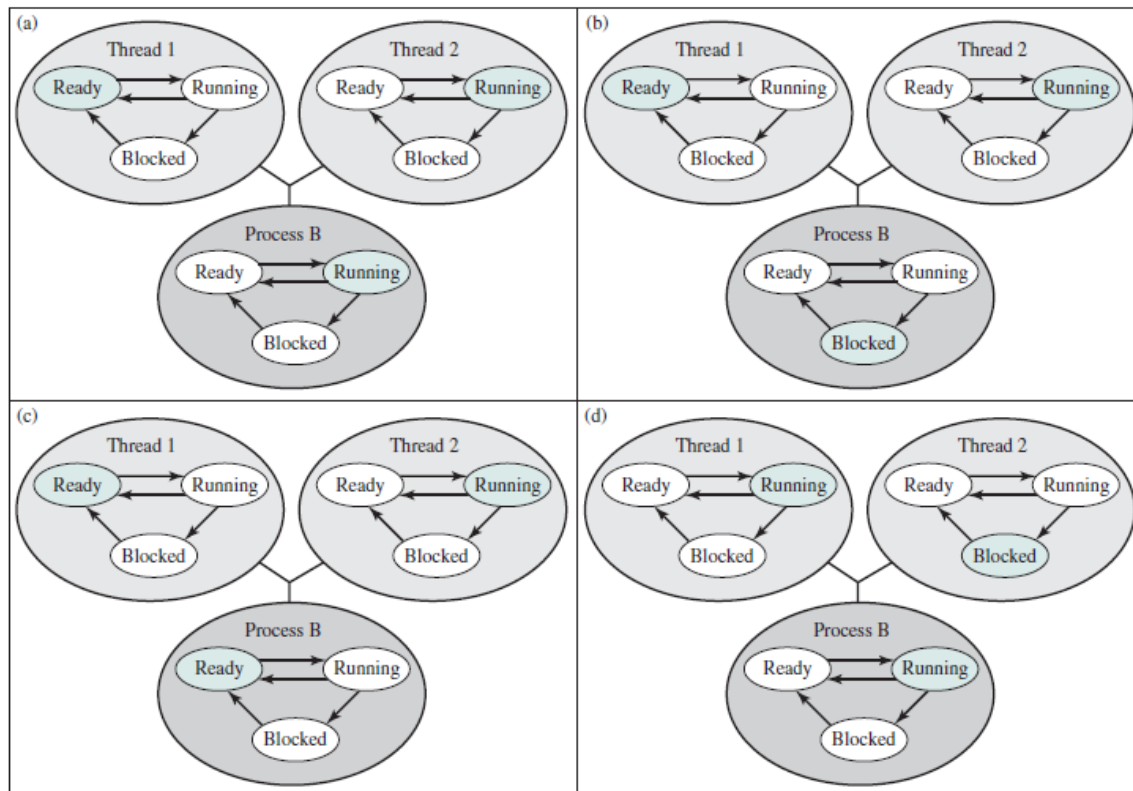
Semua manajemen thread dilakukan oleh aplikasi dan kernel tidak menyadari adanya thread (Gambar 6.8a). Setiap aplikasi dapat diprogram dengan multithreaded menggunakan thread library, yang merupakan sebuah paket dari rutin-rutin untuk manajemen ULT.

Thread library berisi kode untuk membuat dan menghancurkan thread, untuk melewatkan pesan dan data antar thread, untuk penjadwalan eksekusi thread dan untuk menyimpan dan mengembalikan thread contexts.

Sebuah aplikasi dimulai dengan sebuah single thread dan mulai berjalan dengan thread tersebut. Aplikasi ini dan threadnya dialokasikan ke sebuah proses tunggal yang dikelola oleh kernel. Pada saat aplikasi tersebut running (proses berada pada running state), aplikasi tersebut akan melahirkan(spwan) sebuah thread baru untuk berjalan dalam proses yang sama. Spawning dilakukan dengan memanggil spawn utility pada thread library. Control diberikan ke utility tersebut oleh sebuah procedure call.

Thread library membuat sebuah struktur data untuk thread baru dan kemudian memberikan control ke salah satu thread dalam proses yang berada pada ready state, menggunakan algoritma penjadwalan.

Pada saat control diberikan ke library, context dari thread saat ini disimpan, dan kemudian control diberikan dari library ke sebuah thread, context thread tersebut di kembalikan (resotre). Context terdiri dari isi user register, program counter dan stack pointers.



**Gambar 6.9 Hubungan ULT state dan proses state**

Gambar 6.9 diatas merupakan contoh hubungan antara penjadwalan thread dan penjadwalan proses. Misalkan proses B eksekusi thread 2, state proses dan 2 ULT merupakan bagian dari proses (Gambar 6.9a). Berikut ini merupakan kejadian yang mungkin terjadi.

### 1. Gambar 6.9B

Aplikasi mengeksekusi pada thread 2 yang membuat sebuah system call memblock B. sebagai contoh adalah system call I/O. hal ini menyebabkan kontrol untuk melakukan transfer ke kernel. Kernel memberikan tindakan I/O, menempatkan proses B dalam block state dan men-switch ke proses lain. Sedangkan struktur data di kelola oleh thread library, thread 2 dari proses B tetap berada pada running state.

**Note:** Thread 2 tidak benar-benar berjalan (dieksekusi oleh processor), tetapi sedang berjalan pada running state oleh thread library.

## 2. Gambar 6.9C

Sebuah clock interrupt memberikan control ke kernel dan kernel memastikan bahwa proses yang sedang berjalan (proses B) telah habis time slice nya. Kernel menempatkan proses B dalam Ready state dan men-switch ke proses lain. Dalam artian, termasuk struktur data yang dikelola oleh thread library, thread 2 proses B tetap berada pada Running state.

## 3. Gambar 6.9D

Thread 2 sudah mencapai sebuah titik dimana dia memerlukan beberapa aksi yang dilakukan oleh thread 1 dari proses B. thread 2 masuk ke block state dan thread 1 transisi dari ready ke running.

Sedangkan proses sendiri tetap berada pada Running state.

## KERNEL-LEVEL THREADS (KLT)

---

Semua manajemen dari thread dilakukan oleh kernel. Pada bagian ini tidak ada kode manajemen thread dalam level aplikasi, dimana sebuah application programming interface (API) untuk fasilitas kernel thread. Windows merupakan contoh dari pendekatan ini.

Gambar 6.8b menggambarkan pendekatan KLT. Kernel mengelola context informasi untuk proses-proses secara keseluruhan dan untuk individu thread dalam proses.

Penjadwalan oleh kernel dilakukan pada basis thread. Ada pendekatan yang dilakukan yaitu:

1. Kernel dapat secara simultan menjadwalkan banyak thread dari proses yang sama pada multiple processor
2. Jika satu thread dalam proses di-blok, kernel dapat menjadwalkan thread lain dari proses yang sama.

## Kombinasi

---

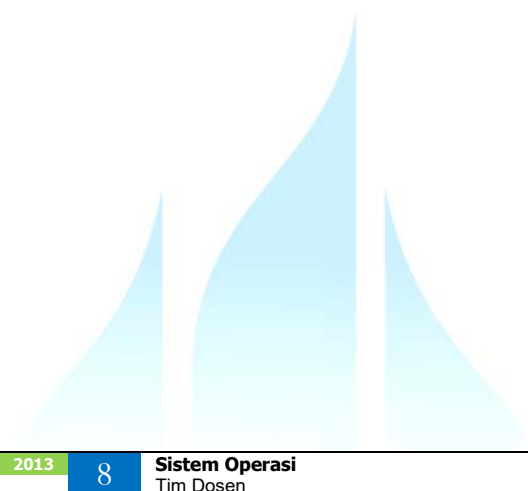
Beberapa OS menyediakan sebuah kombinasi ULT / KLT (Gambar 6.8c). pada sistem kombinasi, penciptaan thread dilakukan secara komplit oleh user, seperti penjadwalan dan sinkronisasi thread dalam sebuah aplikasi.

Banyak ULT dari sebuah aplikasi tunggal dipetakan pada beberapa jumlah KLT. Programmer mungkin

The multiple ULTs from a single application are

mapped onto some (smaller or equal) number of KLTs. The programmer may mengatur jumlah KLT untuk sebuah aplikasi tertentu dan processor untuk mencapai hasil terbaik.

Pada pendekatan ini, multiple threads dalam aplikasi yang sama dapat berjalan parallel pada multiple processors, dan sebuah blocking system call tidak perlu mem-block seluruh proses.



# Daftar Pustaka

1. Stalling.W, *Operating System Internals and Design Principle Seventh Edition*, Prentice hall, 2012
2. Silberschatz. A, Galvin. P.B, Gagne. Greg, *Operating System Concepts Ninth perEdition*, Jhon, Wiley & Sons, 2013





## MODUL PERKULIAHAN

# Sistem Operasi

## Solusi Critical Section

Fakultas  
Ilmu Komputer

Program Studi  
Teknik Informatika

Tatap Muka

07

Kode MK  
87030

Disusun Oleh  
Tim Dosen

### Abstract

Modul ini membahas tentang beberapa algoritma penyelesaian criticalsection

### Kompetensi

Diharapkan mahasiswa mengetahui dan memahami algoritma Critical Section

# Jenis Solusi

Ada dua jenis solusi untuk memecahkan masalah *critical section*, yaitu.

1. **Solusi Perangkat Lunak.**

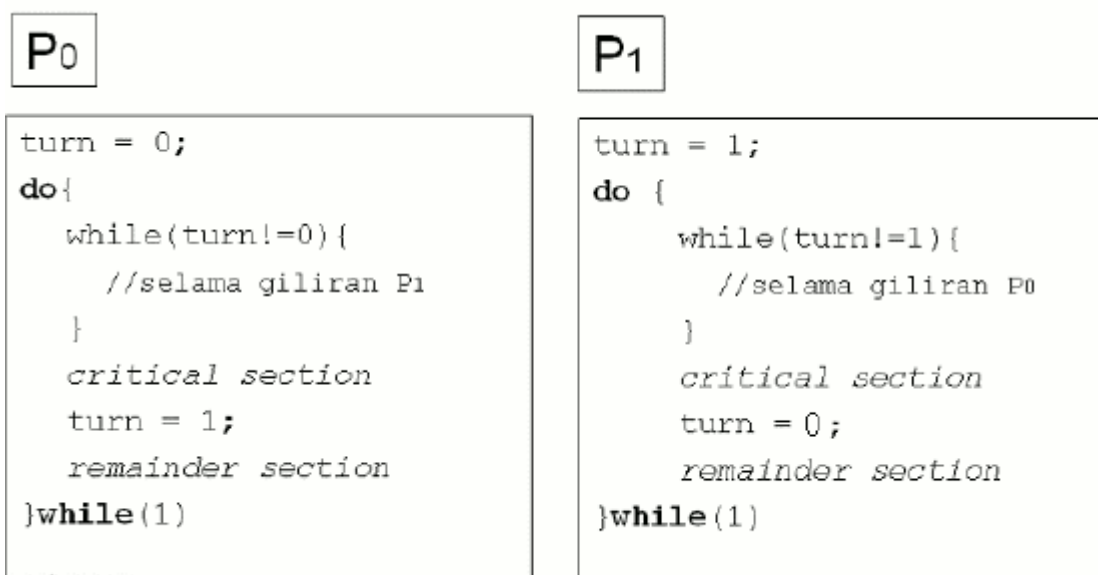
Solusi ini menggunakan algoritma-algoritma untuk mengatasi masalah *critical section*.

2. **Solusi Perangkat Keras.**

Solusi ini tergantung pada beberapa instruksi mesin tertentu, misalnya dengan menon-aktifkan interupsi, mengunci suatu variabel tertentu atau menggunakan instruksi level mesin seperti tes dan set.

## Algoritma 1

Algoritma I mencoba mengatasi masalah *critical section* untuk dua proses. Algoritma ini menerapkan sistem bergilir kepada kedua proses yang ingin mengeksekusi *critical section*, sehingga kedua proses tersebut harus bergantian menggunakan *critical section*.



Gambar 1. Algoritma I

- Algoritma ini menggunakan variabel bernama **turn**, nilai **turn** menentukan proses mana yang boleh memasuki critical section dan mengakses data yang di-sharing.
- Pada awalnya variabel turn diinisialisasi 0, artinya P0 yang boleh mengakses critical section. Jika  $turn = 0$  dan P0 ingin menggunakan critical section, maka ia dapat mengakses critical section-nya.
- Setelah selesai mengeksekusi critical section, P0 akan mengubah turn menjadi 1, yang artinya giliran P1 tiba dan P1 diperbolehkan mengakses critical section.
- Ketika  $turn = 1$  dan P0 ingin menggunakan critical section, maka P0 harus menunggu sampai P1 selesai menggunakan critical section dan mengubah turn menjadi 0.

Ketika suatu proses sedang menunggu, proses tersebut masuk ke dalam *loop*, dimana ia harus terus-menerus mengecek variabel *turn* sampai berubah menjadi gilirannya. Proses menunggu ini disebut *busy waiting*. Sebenarnya *busy waiting* mesti dihindari karena proses ini menggunakan *CPU*. Namun untuk kasus ini, penggunaan *busy waiting* diijinkan karena biasanya proses menunggu hanya berlangsung dalam waktu yang singkat.

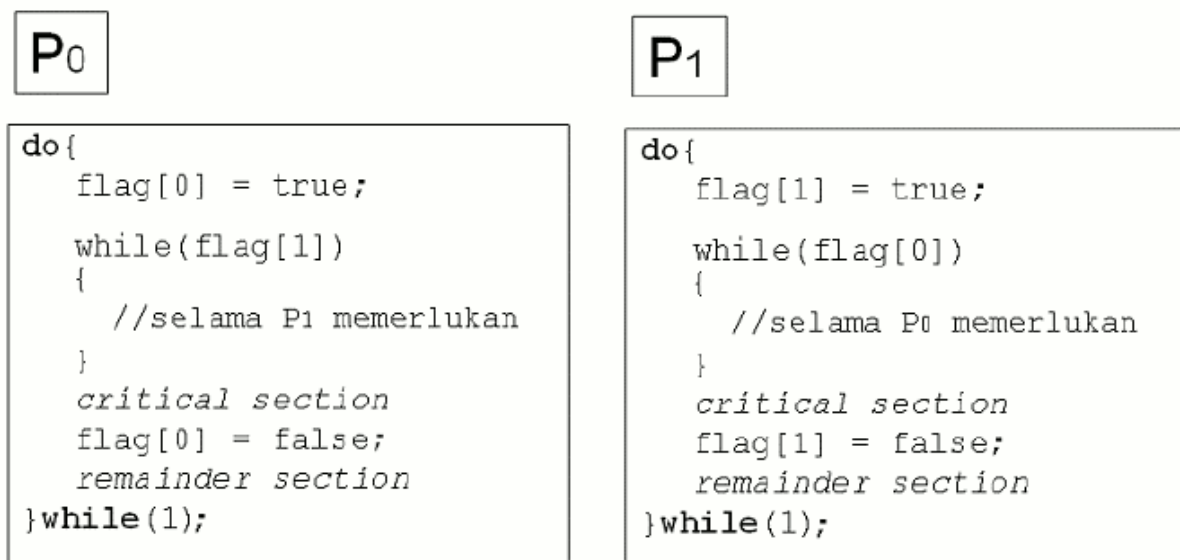
Pada algoritma ini masalah muncul ketika ada proses yang mendapat giliran memasuki *critical section* tapi tidak menggunakan gilirannya sementara proses yang lain ingin mengakses *critical section*. Misalkan ketika  $turn = 1$  dan P1 tidak menggunakan gilirannya maka *turn* tidak berubah dan tetap 1. Kemudian P0 ingin menggunakan *critical section*, maka ia harus menunggu sampai P1 menggunakan *critical section* dan mengubah turn menjadi 0. Kondisi ini tidak memenuhi syarat *progress* karena P0 tidak dapat memasuki *critical section* padahal saat itu tidak ada yang menggunakan *critical section* dan ia harus menunggu P1 mengeksekusi non- *critical section* –nya sampai kembali memasuki *critical section*. Kondisi ini juga tidak memenuhi syarat *bounded waiting* karena jika pada gilirannya P1 mengakses *critical section* tapi P1 selesai mengeksekusi semua kode dan *terminate*, maka tidak ada jaminan P0 dapat mengakses *critical section* dan P0-pun harus menunggu selamanya



# Algoritma 2

Algoritma II juga mencoba memecahkan masalah *critical section* untuk dua proses. Algoritma ini mengantisipasi masalah yang muncul pada algoritma I dengan mengubah penggunaan variabel *turn* dengan variabel *flag*.

Variabel *flag* menyimpan kondisi proses mana yang boleh masuk *critical section*. Proses yang membutuhkan akses ke *critical section* akan memberikan nilai *flag*-nya *true*. Sedangkan proses yang tidak membutuhkan *critical section* akan men-set nilai *flag*-nya bernilai *false*.



**Gambar 2. Algoritma II**

Suatu proses diperbolehkan mengakses *critical section* apabila proses lain tidak membutuhkan *critical section* atau flag proses lain bernilai *false*. Tetapi apabila proses lain membutuhkan *critical section* (ditunjukkan dengan nilai *flag*-nya *true*), maka proses tersebut harus menunggu dan "mempersilakan" proses lain menggunakan *critical section*-nya. Disini terlihat bahwa sebelum memasuki *critical section* suatu proses melihat proses lain terlebih dahulu (melalui *flag*-nya), apakah proses lain membutuhkan *critical section* atau tidak.

Awalnya *flag* untuk kedua proses diinisialisai bernilai *false*, yang artinya kedua proses tersebut tidak membutuhkan *critical section*. Jika P<sub>0</sub> ingin mengakses *critical section*, ia akan mengubah *flag*[0] menjadi *true*. Kemudian P<sub>0</sub> akan mengecek apakah P<sub>1</sub> juga

membutuhkan *critical section*, jika *flag[1]* bernilai *false* maka P0 akan menggunakan *critical section*. Namun jika *flag[1]* bernilai *true* maka P0 harus menunggu P1 menggunakan *critical section* dan mengubah *flag[1]* menjadi *false*.

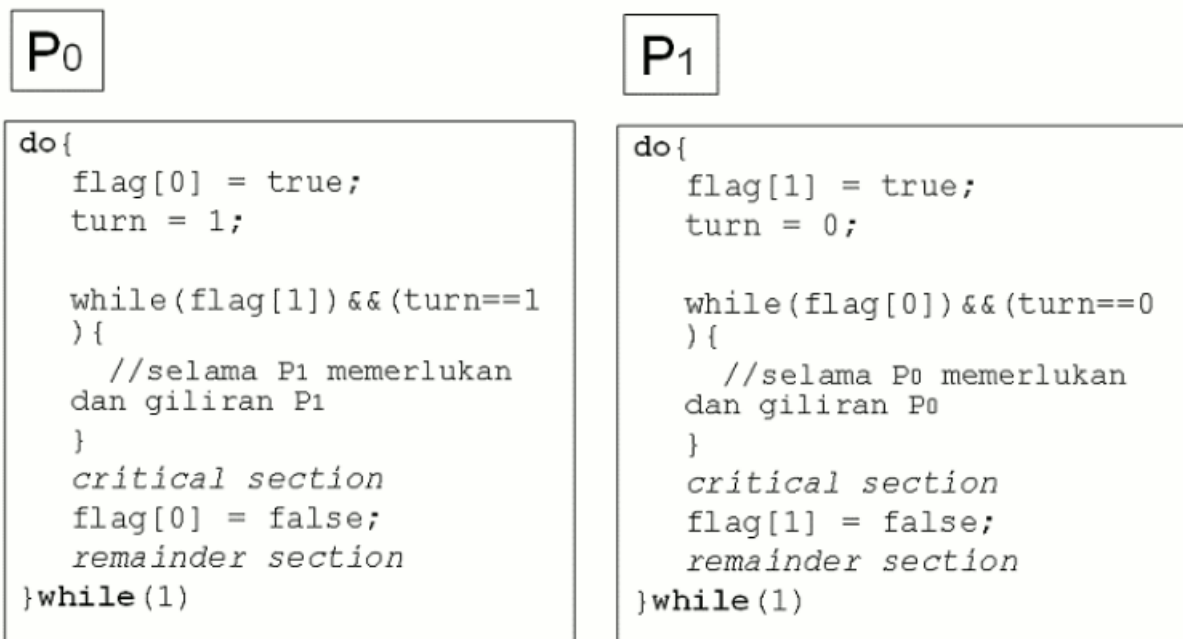
Pada algoritma ini masalah muncul ketika kedua proses secara bersamaan menginginkan *critical section*, kedua proses tersebut akan men- set masing-masing *flag*-nya menjadi *true*. P0 men-set *flag[0] = true*, P1 men-set *flag[1] = true*. Kemudian P0 akan mengecek apakah P1 membutuhkan *critical section*. P0 akan melihat bahwa *flag[1] = true*, maka P0 akan menunggu sampai P1 selesai menggunakan *critical section*. Namun pada saat bersamaan, P1 juga akan mengecek apakah P0 membutuhkan *critical section* atau tidak, ia akan melihat bahwa *flag[0] = true*, maka P1 juga akan menunggu P0 selesai menggunakan *critical section*-nya. Kondisi ini menyebabkan kedua proses yang membutuhkan *critical section* tersebut akan saling menunggu dan "saling mempersilahkan" proses lain untuk mengakses *critical section*, akibatnya malah tidak ada yang mengakses *critical section*. Kondisi ini menunjukkan bahwa Algoritma II tidak memenuhi syarat progress dan syarat bounded waiting, karena kondisi ini akan terus bertahan dan kedua proses harus menunggu selamanya untuk dapat mengakses *critical section*.



# Algoritma III

Algoritma III ditemukan oleh G.L. Petterson pada tahun 1981 dan dikenal juga sebagai Algoritma Petterson. Petterson menemukan cara yang sederhana untuk mengatur proses agar memenuhi *mutual exclusion*. Algoritma ini adalah solusi untuk memecahkan masalah *critical section* pada dua proses.

Ide dari algoritma ini adalah menggabungkan variabel yang di-*sharing* pada Algoritma I dan Algoritma II, yaitu variabel *turn* dan variabel *flag*. Sama seperti pada Algoritma I dan II, variabel *turn* menunjukkan giliran proses mana yang diperbolehkan memasuki *critical section* dan variabel *flag* menunjukkan apakah suatu proses membutuhkan akses ke *critical section* atau tidak.



Gambar 3. Algoritma III

Awalnya *flag* untuk kedua proses diinisialisasi bernilai *false*, yang artinya kedua proses tersebut tidak membutuhkan akses ke *critical section*. Kemudian jika suatu proses ingin memasuki *critical section*, ia akan mengubah *flag*-nya menjadi *true* (memberikan tanda bahwa ia butuh *critical section*) lalu proses tersebut memberikan *turn* kepada lawannya. Jika lawannya tidak menginginkan *critical section* (*flag*-nya *false*), maka proses tersebut dapat menggunakan *critical section*, dan setelah selesai menggunakan *critical section* ia akan mengubah *flag*-nya menjadi *false*. Tetapi apabila proses lawannya juga menginginkan *critical section* maka proses lawan-lah yang dapat memasuki *critical section*, dan proses

tersebut harus menunggu sampai proses lawan menyelesaikan *critical section* dan mengubah *flag*-nya menjadi *false*.

Misalkan ketika P0 membutuhkan *critical section*, maka P0 akan mengubah  $flag[0] = true$ , lalu P0 mengubah  $turn = 1$ . Jika P1 mempunyai  $flag[1] = false$ , (berapapun nilai  $turn$ ) maka P0 yang dapat mengakses *critical section*. Namun apabila P1 juga membutuhkan *critical section*, karena  $flag[1] = true$  dan  $turn = 1$ , maka P1 yang dapat memasuki *critical section* dan P0 harus menunggu sampai P1 menyelesaikan *critical section* dan mengubah  $flag[1] = false$ , setelah itu barulah P0 dapat mengakses *critical section*.

Bagaimana bila kedua proses membutuhkan *critical section* secara bersamaan? Proses mana yang dapat mengakses *critical section* terlebih dahulu? Apabila kedua proses (P0 dan P1) datang bersamaan, kedua proses akan menset masing-masing *flag* menjadi *true* ( $flag[0] = true$  dan  $flag[1] = true$ ), dalam kondisi ini P0 dapat mengubah  $turn = 1$  dan P1 juga dapat mengubah  $turn = 0$ . Proses yang dapat mengakses *critical section* terlebih dahulu adalah proses yang terlebih dahulu mengubah  $turn$  menjadi  $turn$  lawannya. Misalkan P0 terlebih dahulu mengubah  $turn = 1$ , lalu P1 akan mengubah  $turn = 0$ , karena  $turn$  yang terakhir adalah 0 maka P0-lah yang dapat mengakses *critical section* terlebih dahulu dan P1 harus menunggu.

Algoritma III memenuhi ketiga syarat yang dibutuhkan. Syarat *progress* dan *bounded waiting* yang tidak dipenuhi pada Algoritma I dan II dapat dipenuhi oleh algoritma ini karena ketika ada proses yang ingin mengakses *critical section* dan tidak ada yang menggunakan *critical section* maka dapat dipastikan ada proses yang bisa menggunakan *critical section*, dan proses tidak perlu menunggu selamanya untuk dapat masuk ke *critical section*.



# Daftar Pustaka

1. Stalling.W, *Operating System Internals and Design Principle Seventh Edition*, Prentice hall, 2012
2. Silberschatz. A, Galvin. P.B, Gagne. Greg, *Operating System Concepts Ninth perEdition*, Jhon, Wiley & Sons, 2013
3. <http://bebas.vlsm.org/v06/Kuliah/SistemOperasi/BUKU/>





## MODUL PERKULIAHAN

# Sistem Operasi

## Sinkronisasi Proses

Fakultas  
Ilmu Komputer

Program Studi  
Teknik Informatika

Tatap Muka

08

Kode MK  
87030

Disusun Oleh  
Tim Dosen

### Abstract

Modul ini membahas tentang bagaimana proses-proses berkomunikasi dan melakukan sinkronisasi

### Kompetensi

Diharapkan mahasiswa mengetahui konsep mutual exclusion, critical section, starvation dan deadlock

# Defenisi

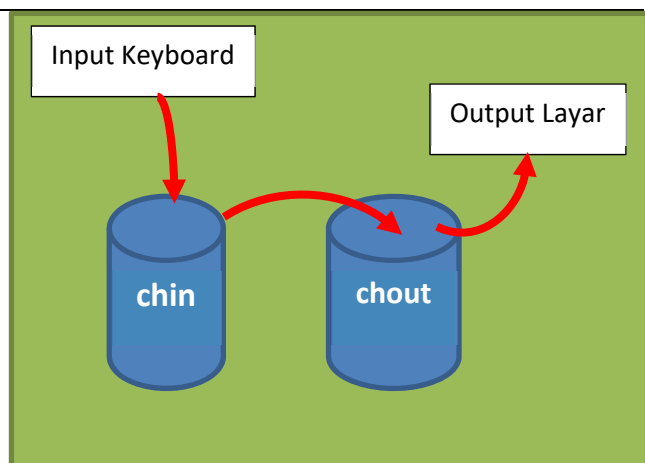
Desain dari OS pada umumnya terkonsentrasi kepada manajemen proses dan thread pada:

- **Multiprogramming** → manajemen dari banyak proses pada sebuah system uni uniprocessor
- **Multiprocessing** → manajemen banyak proses pada multiprocessor
- **Distributed processing** → manajemen banyak proses yang dijalankan pada banyak system computer terdistribusi

Dasar dari hal-hal diatas dan merupakan desain dasar OS adalah concurrency atau konkurensi. Konkurensi mencakup berbagai masalah desain seperti komunikasi antar proses, sharing dan kompetisi untuk sumber daya (seperti memory, file dan akses ke I/O), sinkronisasi aktifitas dari banyak proses dan alokasi waktu processor untuk setiap proses.

## Contoh sederhana

```
void echo()  
{  
    chin = getchar();  
    chout = chin;  
    putchar(chout);  
}
```



### Keterangan

- Prosedur ini merupakan sub program yang akan menyediakan sebuah karakter yaitu **echo procedure**
- Input diperoleh dari keyboard, dimana 1 keystroke pada satu waktu
- Setiap input karakter disimpan pada variable **chin**
- Ditransfer ke variable **chout**
- Tampilkan ke layar
- Setiap program dapat memanggil prosedur ini berkali-kali untuk menerima input user dan menampilkannya ke layar user.

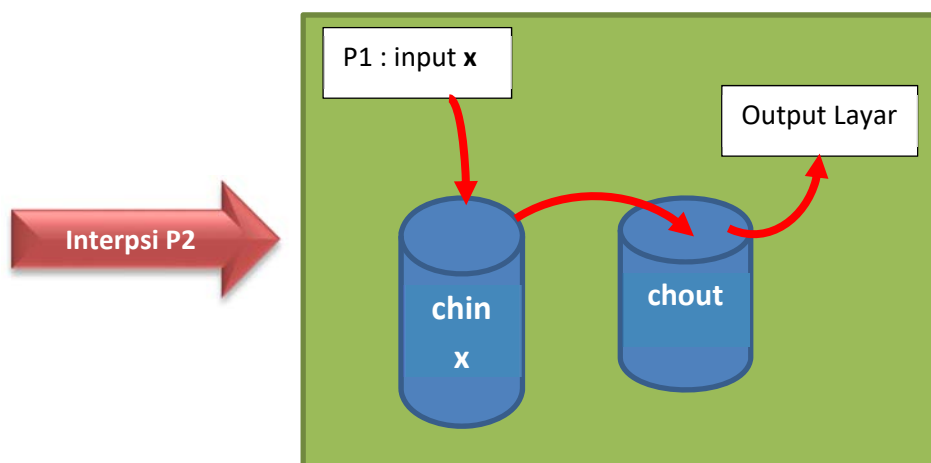
## Single-processor multiprogramming system → single user

User dapat melompat dari satu aplikasi ke aplikasi lain dan setiap aplikasi menggunakan keyboard yang sama untuk input dan layar yang sama untuk output.

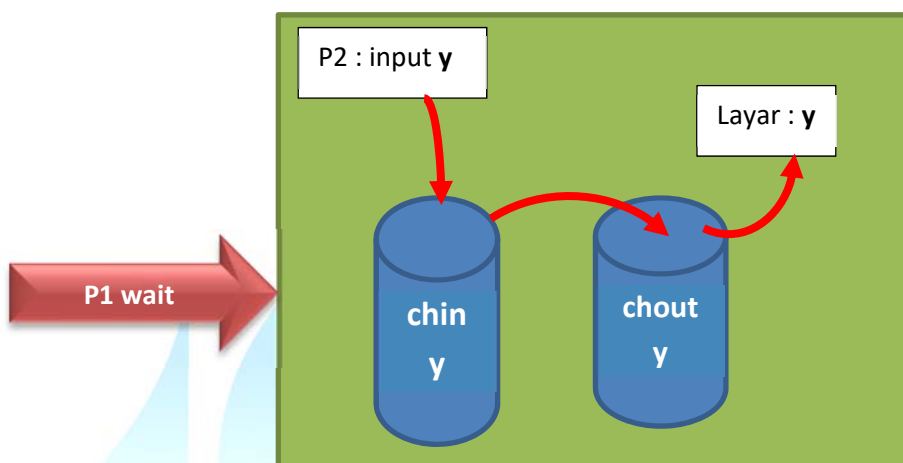
Karena setiap aplikasi menggunakan prosedur **echo**, maka menjadi sebuah procedure yang dishare dan di-load kedalam memory global untuk semua aplikasi. Tetapi hanya satu copy dari prosedur **echo** yang digunakan, untuk menghemat ruang memory.

Sharing main memory antara proses berfungsi untuk mengizinkan efisiensi dan interaksi yang lebih dekat antar proses. Tetapi sharing sendiri dapat menimbulkan masalah, seperti berikut:

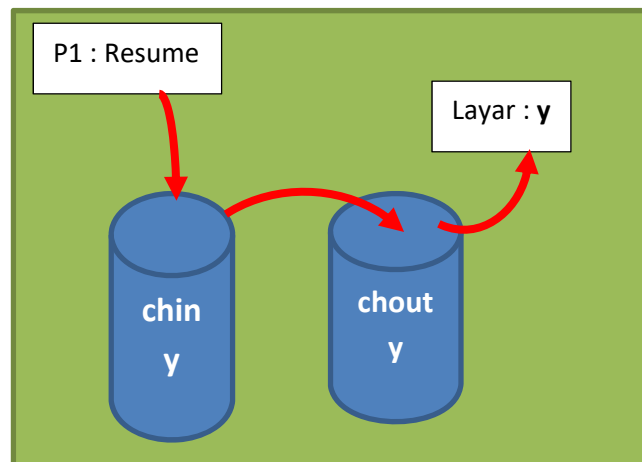
1. Process P1 meminta echo procedure dan di-interupsi oleh proses P2 setelah getchar dan menyimpan ke chin . Misal, input adakah **x** , **x** disimpan pada variable **chin** .



2. Process P2 diaktifkan dan meminta echo procedure, dengan menginputkan dan menampilkan karakter tunggal **y** pada layar.



3. Process P1 dilanjutkan. Pada saat ini, nilai x pada chin sudah ditimpa dan hilang. Karena chin berisi y, ditransfer ke chout dan ditampilkan y.



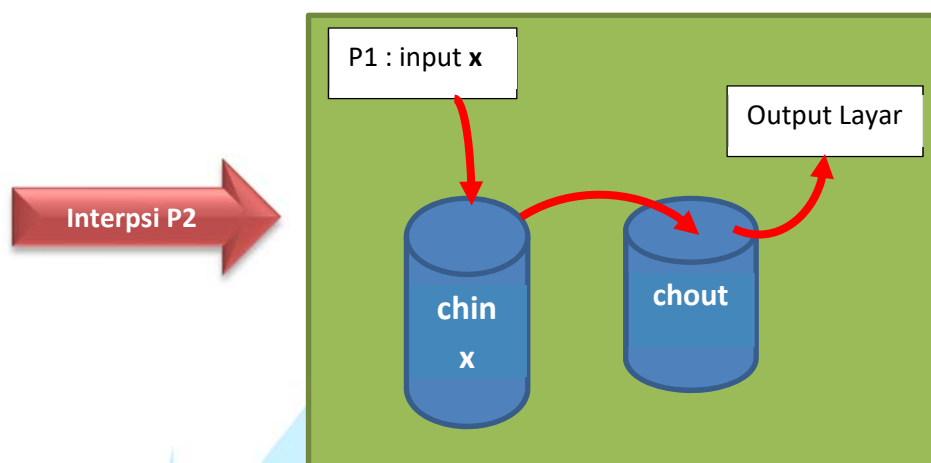
Karakter pertama hilang dan karakter kedua ditampilkan 2 kali.

**Inti permasalahan adalah shared global variable, chin .**

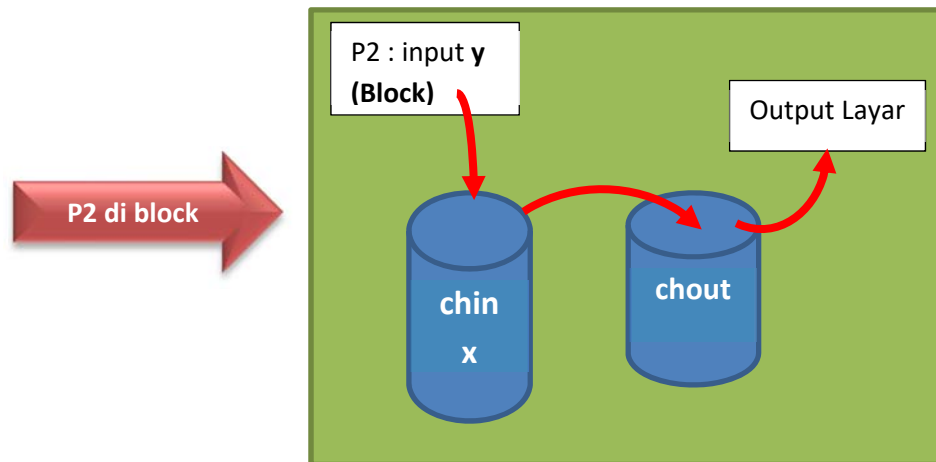
Banyak proses memiliki akses ke variable ini. Jika satu proses melakukan update global variabel dan kemudian di interrupt, maka proses lain akan merubah variable sebelum proses pertama menggunakannya.

**Solusi :** hanya satu proses yang diizinkan mengakses procedure pada satu waktu.

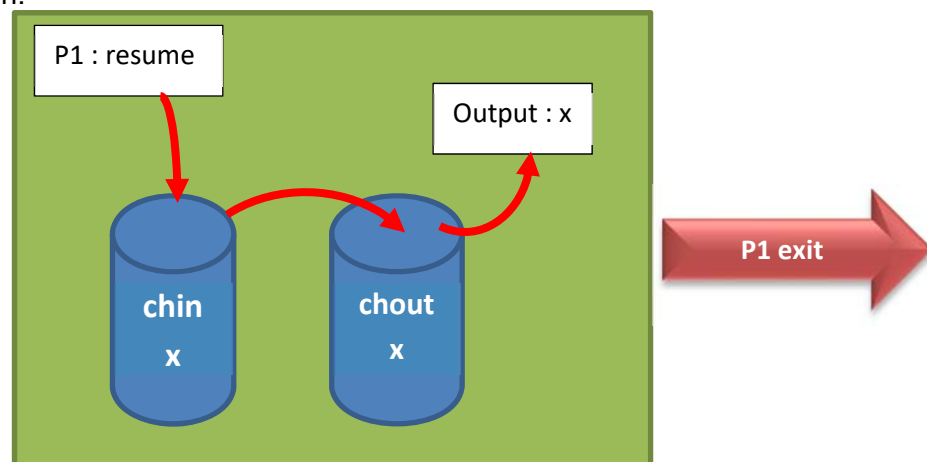
1. Process P1 meminta echo procedure dan diinterupsi setelah selesai fungsi input. Pada bagian ini, diinput karakter, x, disimpan pada variable chin .



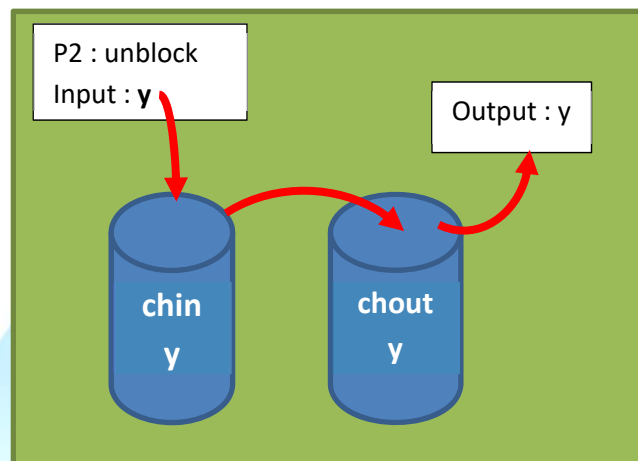
2. Process P2 diaktifkan dan meminta echo procedure. Namun , karena P1 masih berada pada echo procedure, P2 akan di-block untuk masuk ke prosedur. P2 di tangguhkan dan menunggu sampai prosedur echo tersedia.



3. Pada saat process P1 dilanjutkan dan menyelesaikan eksekusi echo, maka karakter x ditampilkan.



4. Pada saat P1 exits echo , maka block P2 akan dihapus. Ketika P2 di-resume, echo procedure sukses diperoleh.



## Multiprocessor multiprogramming system

---

Pada sistem multiprocessor, masalah yang muncul adalah sama yaitu bagaimana melakukan proteksi pada sumber daya yang di bagi.

### Contoh:

Dalam hal ini tidak ada mekanisme untuk melakukan pengontrolan akses ke global variable yang dibagi.

1. Proses P1 dan P2 berjalan pada prosessor yang terpisah. Kedua proses meminta echo procedure.
2. Yang terjadi adalah sebagai berikut; event-event pada line yang sama dieksekusi secara paralel:

Process P1	Process P2
•	•
<code>chin = getchar();</code>	•
•	<code>chin = getchar();</code>
<code>chout = chin;</code>	<code>chout = chin;</code>
<code>putchar(chout);</code>	•
•	<code>putchar(chout);</code>
•	•

### Hasil

=====

Input karakter untuk P1 hilang sebelum ditampilkan, dan input karater untuk P2 ditampilkan oleh kedua proses yaitu P1 dan P2.

Jika ditambahkan aturan bahwa hanya satu proses pada satu waktu yang berada pada prosedur echo, maka yang terjadi adalah sebagai berikut:

1. Proses P1 dan P2 dieksekusi, masing-masing pada processor yang berbeda. P1 meminta echo procedure.
2. Pada saat P1 berada dalam echo procedure, P2 juga meminta echo . karena P1 tetap berada dalam echo procedure, P2 diblok memasuki echo procedure. Sehingga P2 akan menunggu ketersediaan echo procedure.
3. Pada saat proses P1 selesai mengeksekusi echo , keluar dari procedure. Setelah P1 keluar dari echo procedure, P2 langsung dilanjutkan dan memulai eksekusi echo.

## KESIMPULAN

Pada uniprocessor system.

Masalah:

- Jika terjadi sebuah interrupt, menyebabkan dihentikannya eksekusi intruksi dari sebuah proses.

Pada multiprocessor system

Masalah :

- Sama dengan uniprocessor, disebabkan 2 proses yang dieksekusi secara bersamaan dan keduanya mencoba untuk mengakses variabel global yang sama.

Solusi : sama

Kontrol akses ke shared resource.

## Kompetisi antar proses untuk sumber daya

Proses yang terjadi secara bersamaan mendatangkan konflik ketika berkompetisi untuk menggunakan sumber daya yang sama

Contoh:

2 atau lebih proses memerlukan akses ke sebuah sumber daya pada saat eksekusi. Masing masing proses tidak menyadari adanya proses lain, dan masing masing tidak saling terpengaruh oleh eksekusi proses lain. Contoh dari sumber daya adalah perangkat I/O, memori, waktu processor dan clock.

Dalam hal ini tidak ada pertukaran informasi antar proses-proses yang berebut sumber daya. Namun, eksekusi satu proses akan mempengaruhi perilaku dari proses yang berkompetisi.

Jika 2 proses ingin mengakses sebuah sumber daya tunggal, kemudian satu proses akan dialokasikan sumber daya tersebut oleh OS dan yang lain harus menunggu. Sehingga proses yang aksesnya ditolak akan melambat.

Dalam kasus yang lebih ekstrim, proses yang diblok bisa jadi tidak pernah mendapatkan akses ke sumber daya dan tidak pernah di terminasi secara sukses.

Dalam hal kompetisi proses ada 3 masalah yang akan dihadapi yaitu:

- **mutual exclusion**

- ⌘ Misalkan 2 atau lebih proses membutuhkan akses ke sebuah sumber daya tunggal yang nonsharable, seperti printer. Selama eksekusi setiap proses akan mengirimkan perintah-perintah ke perangkat I/O, menerima informasi status, mengirim data, dan/atau menerima data.
- ⌘ Maka sumber daya seperti ini merupakan sebuah **critical resource**, dan bagian program yang menggunakannya merupakan sebuah **critical section** dari program.

```
do {  
    entry section  
    critical section  
    exit section  
    remainder section  
} while (true);
```

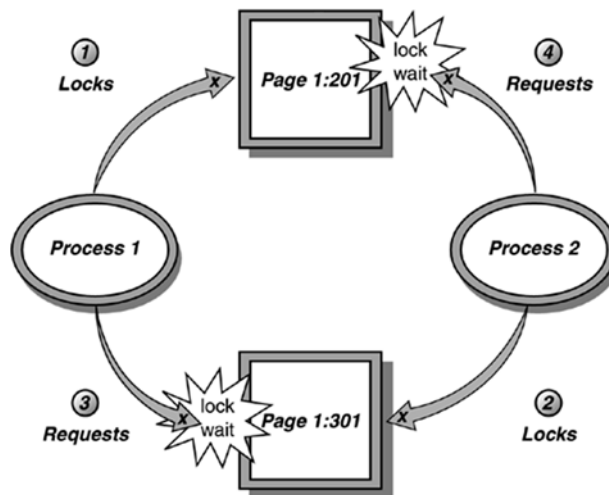
General structure of a typical process  $P_i$ .

- ⌘ Hanya satu program pada satu waktu yang diizinkan dalam critical season-nya.
- ⌘ Pelaksanaan dari mutual exclusion dapat menyebabkan 2 masalah yaitu **deadlock** dan **starvation**.

- **Deadlock**

- ⌘ 2 proses P1 dan P2 dan 2 sumber daya R1 dan R2. Dimana masing-masing proses harus mengakses kedua sumber daya untuk melaksanakan bagian dari fungsinya. Situasi yang mungkin terjadi adalah OS memberikan R1 ke P2 dan R2 ke P1. Setiap proses menunggu salah satu dari 2 sumber daya. Setiap proses tidak akan melepaskan sumber daya yang dimiliki sampai diperoleh sumber daya lain dan untuk melaksanakan fungsi membutuhkan kedua proses. Kedua proses ini berada dalam kondisi deadlock.

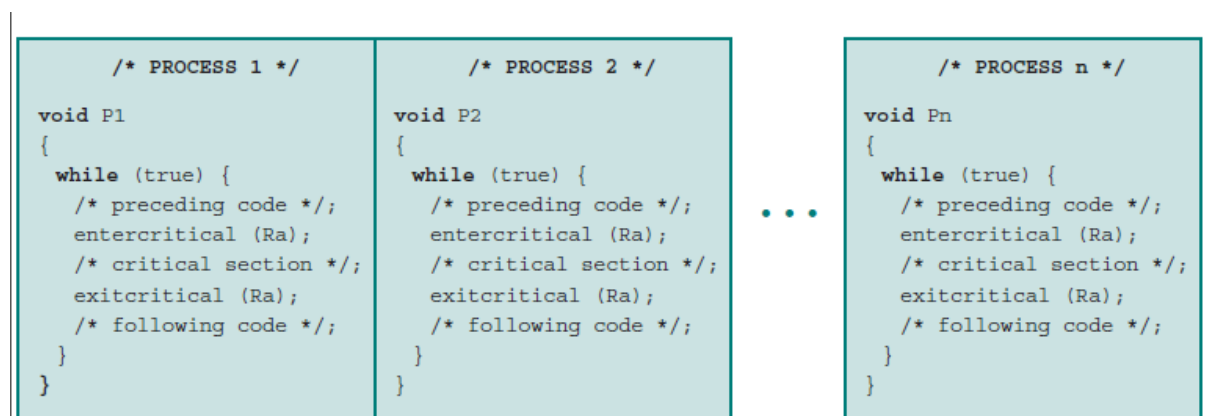




- **Starvation**

3 proses (P1, P2, P3) masing-masing memerlukan akses secara periodik ke sumber daya R. dan P2 dan P3 di-delay. Menunggu sumber daya tersebut. Pada saat P1 keluar dari critical section, salah satu proses P2 atau P3 harus diizinkan untuk akses ke R. Misalkan OS memberikan akses ke P3 dan kemudian P1 harus mengakses R lagi sebelum P3 menyelesaikan critical sectionnya. Jika OS memberikan akses ke P1 setelah P3 selesai, dan kemudian bergantian memberikan akses ke P1 dan P3, maka P2 kemungkinan akan ditolak aksesnya ke sumber daya tanpa batas, walaupun tidak ada situasi deadlock.

## Ilustrasi Mekanisme Mutual Exclusion



Ada  $n$  proses yang dieksekusi secara bersamaan. Setiap proses melibatkan :

1. Sebuah critical section yang beroperasi pada beberapa sumber daya Ra
2. Kode tambahan yang mendahului dan mengikuti critical section yang tidak melibatkan akses ke Ra.

Karena semua proses mengakses sumber daya yang sama Ra, dimana hanya ada 1 proses pada satu waktu dalam critical section nya.

Untuk melaksanakan mutual exclusion, maka disediakan 2 fungsi yaitu entercritical dan exitcritical. Jika proses P1 mencoba masuk ke critical section-nya pada saat proses P2 berada dalam critical section nya, untuk sumber daya yang sama, maka proses P1 harus menunggu.



# Daftar Pustaka

1. Stalling.W, *Operating System Internals and Design Principle Seventh Edition*, Prentice hall, 2012
2. Silberschatz. A, Galvin. P.B, Gagne. Greg, *Operating System Concepts Ninth perEdition*, Jhon, Wiley & Sons, 2013



## MODUL PERKULIAHAN

# Sistem Operasi

## Investigasi Sinkronisasi Proses

Fakultas  
Ilmu Komputer

Program Studi  
Teknik Informatika

Tatap Muka

09

Kode MK  
87030

Disusun Oleh  
Tim Dosen

### Abstract

Modul ini membahas tentang bagaimana proses-proses berkomunikasi dan melakukan sinkronisasi menggunakan SIMULATOR YASS

### Kompetensi

Diharapkan mahasiswa mengetahui konsep mutual exclusion, critical section, starvation dan deadlock dengan menggunakan simulator YASS

# Investigating Synchronisation

## Introduction

---

At the end of this lab you should be able to:

1. Show that writing to unprotected shared global memory region can have undesirable side effects when accessed by threads at the same time.
2. Understand shared global memory protection using synchronised threads.
3. Explain how critical regions of code can protect shared global memory areas.
4. Show that memory areas local to threads are unaffected by other threads.

## Processor and OS Simulators

---

The computer architecture tutorials are supported by simulators, which are created to underpin theoretical concepts normally covered during the lectures. The simulators provide visual and animated representation of mechanisms involved and enable the students to observe the hidden inner workings of systems, which would be difficult or impossible to do otherwise. The added advantage of using simulators is that they allow the students to experiment and explore different technological aspects of systems without having to install and configure the real systems.

## Basic Theory

---

Concurrent processes accessing global shared resources at the same time can produce unpredictable side-effects if the resources are unprotected. Computer hardware and operating system can provide support for implementing critical regions of code when globally accessible resources are shared by several concurrently executing threads.



# Lab Exercises - Investigate and Explore

Start the CPU simulator. You now need to create some executable code so that it can be run by the CPU under the control of the OS. In order to create this code, you need to use the compiler which is part of the system simulator. To do this, open the compiler window by selecting the **COMPILER...** button in the current window.

1. In the compiler window, enter the following source code in the compiler source editor area (under **PROGRAM SOURCE** frame title). Make sure your program is exactly the same as the one below (best to use copy and paste for this).

```
program CriticalRegion1
var g integer
sub thread1 as thread
writeln("In thread1")
g = 0
for n = 1 to 20
g = g + 1
next
writeln("thread1 g = ", g)
writeln("Exiting thread1")
end sub
sub thread2 as thread
writeln("In thread2")
g = 0
for n = 1 to 12
g = g + 1
next
writeln("thread2 g = ", g)
writeln("Exiting thread2")
end sub
writeln("In main")
call thread1
call thread2
wait
writeln("Exiting main")
end
```

The above code creates a main program called *CriticalRegion1*. This program creates two threads thread1 and thread2. Each thread increments the value of the global variable **g** in two separate loops.

Work out what two values of  $g$  you would expect to be displayed on the console when the two threads finish?



Compiling and loading the above code:

- Compile the above code using the **COMPILE...** button.
- Load the CPU instructions in memory using the **LOAD IN MEMORY** button.
- Display the console using the **INPUT/OUTPUT...** button in CPU simulator.
- On the console window check the **Stay on top** check box.

Running the above code:

- Enter the OS simulator using the OS 0... button in CPU simulator.
- You should see an entry, titled CriticalRegion1, in the PROGRAM LIST view.
- Create an instance of this program using the NEW PROCESS button.
- Select Round Robin option in the SCHEDULER/Policies view.
- Select 10 ticks from the drop-down list in RR Time Slice frame.
- Make sure the console window is displaying (see above).
- Move the Speed slider to the fastest position.
- Start the scheduler using the START button.

Now, follow the instructions below without any deviations:

2. When the program stops running, make a note of the two displayed values of  $g$ . Are these values what you were expecting? Explain if there are any discrepancies.



3. Change **RR Time Slice** in the OS simulator window to **5 ticks** and repeat the above run. Again, make note of the two values of the variable **g**. Are these different than the values in (2) above? If so, explain why.



4. Modify this program as shown below. The changes are in bold and underlined. Rename the program *CriticalRegion2*.

```
program CriticalRegion2
var g integer
sub thread1 as thread synchronise
writeln("In thread1")
g = 0
for n = 1 to 20
g = g + 1
next
writeln("thread1 g = ", g)
writeln("Exiting thread1")
end sub
sub thread2 as thread synchronise
writeln("In thread2")
g = 0
for n = 1 to 12
g = g + 1
next
writeln("thread2 g = ", g)
writeln("Exiting thread2")
end sub
writeln("In main")
call thread1
call thread2
wait
writeln("Exiting main")
end
```

**NOTE:** The **synchronise** keyword makes sure the thread1 and thread2 code are executed mutually exclusively (i.e. not at the same time).



5. Compile the above program and load in memory as before. Next, run it and carefully observe how the threads behave. Make a note of the two values of variable **g**. Are the results different than those in (2) and (3) above? If so, why?



6. Modify this program for the second time. The new additions are in bold and underlined. Remove the two **synchronise** keywords. Rename it *CriticalRegion3*.

```
program CriticalRegion3
var g integer
sub thread1 as thread
  writeln("In thread1")
enter
  g = 0
  for n = 1 to 20
    g = g + 1
  next
  writeln("thread1 g = ", g)
leave
  writeln("Exiting thread1")
end sub
sub thread2 as thread
  writeln("In thread2")
  enter
  g = 0
  for n = 1 to 12
    g = g + 1
  next
  writeln("thread2 g = ", g)
  leave
  writeln("Exiting thread2")
end sub
writeln("In main")
call thread1
call thread2
wait
writeln("Exiting main")
end
```

NOTE: The **enter** and **leave** keyword pair protect the program code between them. This makes sure the protected code executes exclusively without sharing the CPU with any other thread.

7. Locate the CPU assembly instructions generated for the enter and leave keywords in the compiler's PROGRAM CODE view. You can do this by clicking in the source editor on any of the above keywords. Corresponding CPU instruction will be highlighted. Make a note of this instruction here:
8. Compile the above program and load in memory as before. Next, run it. Make a note of the two values of variable **g**.

9. Modify this program for the third time. The new additions are in bold and underlined. Remove the global variable **g**, enter and leave keywords. Rename it CriticalRegion4.

```
program CriticalRegion4
sub thread1 as thread
var g integer
writeln("In thread1")
g = 0
for n = 1 to 20
g = g + 1
next
writeln("thread1 g = ", g)
writeln("Exiting thread1")
end sub
sub thread2 as thread
var g integer
writeln("In thread2")
g = 0
for n = 1 to 12
g = g + 1
next
writeln("thread2 g = ", g)
writeln("Exiting thread2")
end sub
writeln("In main")
call thread1
call thread2
wait
writeln("Exiting main")
end
```

10. Compile the above program and load in memory as before. Next, run it. Make a note of the two values of variable `g`. How do the new `g` variables differ than the ones in (1), (4) and (6) above?

11. So what have we done so far? To help understand theory better try to answer the following questions. You need to include this in your portfolio, so it is important that you attempt all the questions below. However, you don't need to complete this part during the tutorial session.

- Briefly explain the main purpose of this tutorial as you understand it.
- Why have we chosen to display the same global variable `g` in both threads?
- What popular high-level language uses the keyword **synchronise** (or similar) for the same purpose as the code in (4)?
- Critical regions are often implemented using **semaphores** and **mutexes**. Find out what these are and how they differ. Describe on a separate sheet.
- Some computer architectures have a “test-and-set” CPU instruction for implementing critical regions. Find out how this works and briefly describe on a separate sheet.
- In the absence of any help from hardware and operating system, how would you protect a critical region in your code? Suggest a way of doing it and state how it would differ from the above methods (hint: “busy wait”).

# Daftar Pustaka

1. Stalling.W, *Operating System Internals and Design Principle Seventh Edition*, Prentice hall, 2012
2. Silberschatz. A, Galvin. P.B, Gagne. Greg, *Operating System Concepts Ninth perEdition*, Jhon, Wiley & Sons, 2013
3. <http://www.teach-sim.com/>



## MODUL PERKULIAHAN

# Sistem Operasi

## Solusi Critical Section

Fakultas  
Ilmu Komputer

Program Studi  
Teknik Informatika

Tatap Muka

10

Kode MK  
87030

Disusun Oleh  
Tim Dosen

### Abstract

Modul ini membahas tentang beberapa algoritma penyelesaian criticalsection

### Kompetensi

Diharapkan mahasiswa mengetahui dan memahami algoritma Critical Section

# Jenis Solusi

Ada dua jenis solusi untuk memecahkan masalah *critical section*, yaitu.

1. **Solusi Perangkat Lunak.**

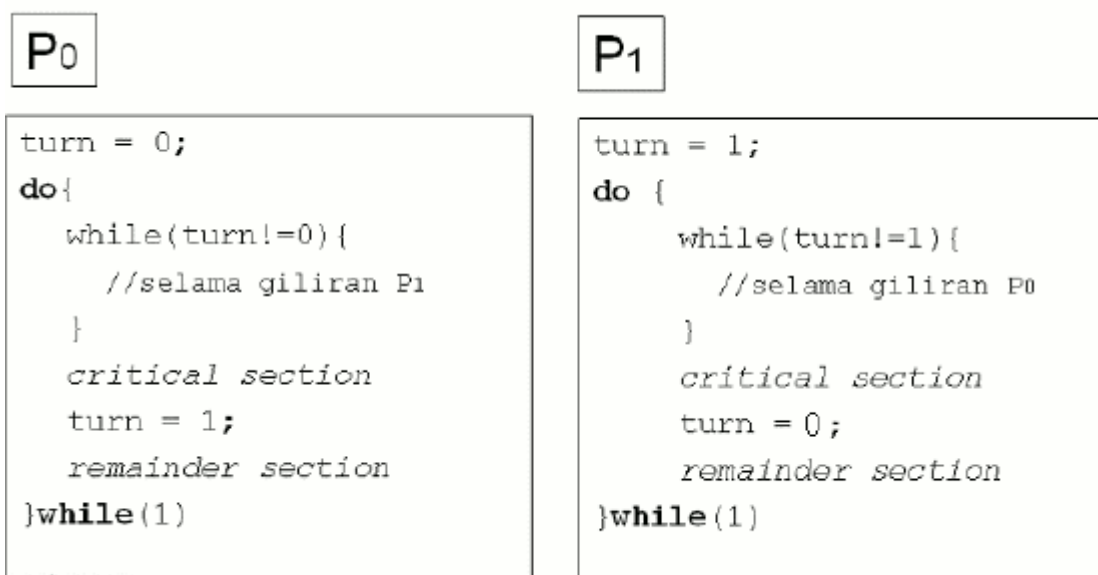
Solusi ini menggunakan algoritma-algoritma untuk mengatasi masalah *critical section*.

2. **Solusi Perangkat Keras.**

Solusi ini tergantung pada beberapa instruksi mesin tertentu, misalnya dengan menon-aktifkan interupsi, mengunci suatu variabel tertentu atau menggunakan instruksi level mesin seperti tes dan set.

## Algoritma 1

Algoritma I mencoba mengatasi masalah *critical section* untuk dua proses. Algoritma ini menerapkan sistem bergilir kepada kedua proses yang ingin mengeksekusi *critical section*, sehingga kedua proses tersebut harus bergantian menggunakan *critical section*.



Gambar 1. Algoritma I

- Algoritma ini menggunakan variabel bernama **turn**, nilai **turn** menentukan proses mana yang boleh memasuki critical section dan mengakses data yang di-sharing.
- Pada awalnya variabel turn diinisialisasi 0, artinya P0 yang boleh mengakses critical section. Jika  $turn = 0$  dan P0 ingin menggunakan critical section, maka ia dapat mengakses critical section-nya.
- Setelah selesai mengeksekusi critical section, P0 akan mengubah turn menjadi 1, yang artinya giliran P1 tiba dan P1 diperbolehkan mengakses critical section.
- Ketika  $turn = 1$  dan P0 ingin menggunakan critical section, maka P0 harus menunggu sampai P1 selesai menggunakan critical section dan mengubah turn menjadi 0.

Ketika suatu proses sedang menunggu, proses tersebut masuk ke dalam *loop*, dimana ia harus terus-menerus mengecek variabel *turn* sampai berubah menjadi gilirannya. Proses menunggu ini disebut *busy waiting*. Sebenarnya *busy waiting* mesti dihindari karena proses ini menggunakan *CPU*. Namun untuk kasus ini, penggunaan *busy waiting* diijinkan karena biasanya proses menunggu hanya berlangsung dalam waktu yang singkat.

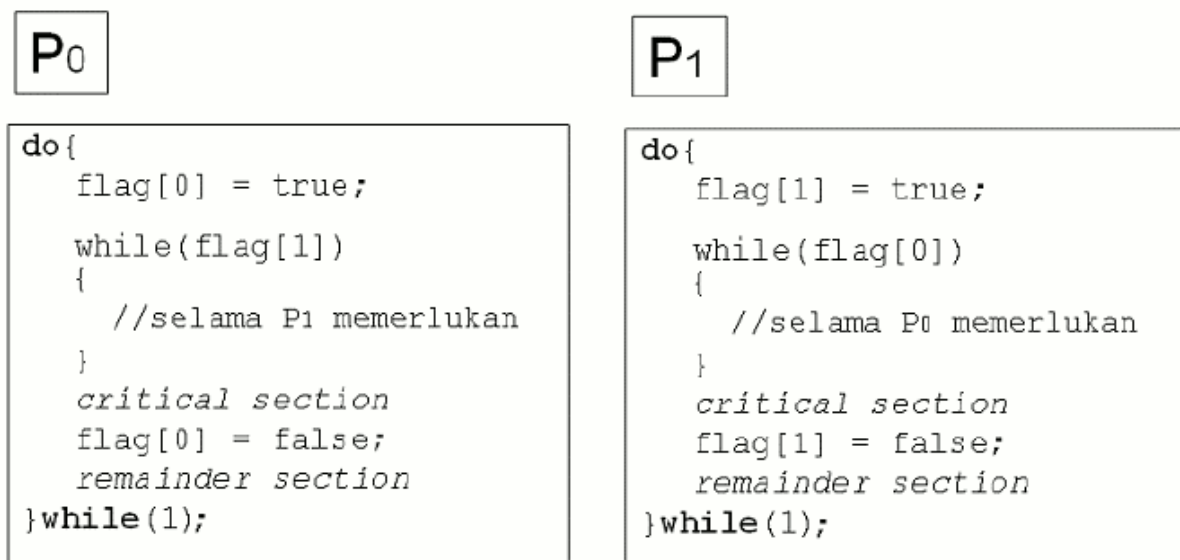
Pada algoritma ini masalah muncul ketika ada proses yang mendapat giliran memasuki *critical section* tapi tidak menggunakan gilirannya sementara proses yang lain ingin mengakses *critical section*. Misalkan ketika  $turn = 1$  dan P1 tidak menggunakan gilirannya maka *turn* tidak berubah dan tetap 1. Kemudian P0 ingin menggunakan *critical section*, maka ia harus menunggu sampai P1 menggunakan *critical section* dan mengubah turn menjadi 0. Kondisi ini tidak memenuhi syarat *progress* karena P0 tidak dapat memasuki *critical section* padahal saat itu tidak ada yang menggunakan *critical section* dan ia harus menunggu P1 mengeksekusi non- *critical section* –nya sampai kembali memasuki *critical section*. Kondisi ini juga tidak memenuhi syarat *bounded waiting* karena jika pada gilirannya P1 mengakses *critical section* tapi P1 selesai mengeksekusi semua kode dan *terminate*, maka tidak ada jaminan P0 dapat mengakses *critical section* dan P0-pun harus menunggu selamanya



## Algoritma 2

Algoritma II juga mencoba memecahkan masalah *critical section* untuk dua proses. Algoritma ini mengantisipasi masalah yang muncul pada algoritma I dengan mengubah penggunaan variabel *turn* dengan variabel *flag*.

Variabel *flag* menyimpan kondisi proses mana yang boleh masuk *critical section*. Proses yang membutuhkan akses ke *critical section* akan memberikan nilai *flag*-nya *true*. Sedangkan proses yang tidak membutuhkan *critical section* akan men-set nilai *flag*-nya bernilai *false*.



Gambar 2. Algoritma II

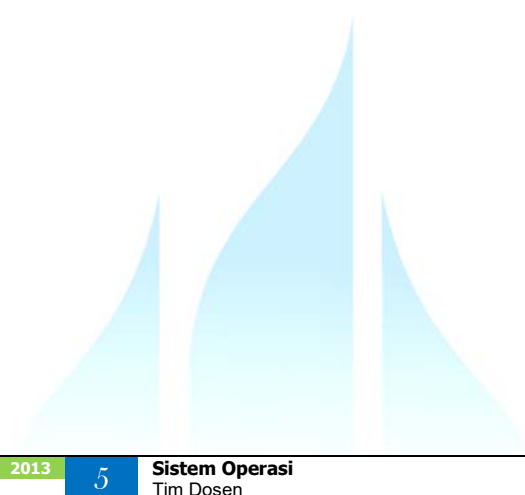
Suatu proses diperbolehkan mengakses *critical section* apabila proses lain tidak membutuhkan *critical section* atau flag proses lain bernilai *false*. Tetapi apabila proses lain membutuhkan *critical section* (ditunjukkan dengan nilai *flag*-nya *true*), maka proses tersebut harus menunggu dan "mempersilakan" proses lain menggunakan *critical section*-nya. Disini terlihat bahwa sebelum memasuki *critical section* suatu proses melihat proses lain terlebih dahulu (melalui *flag*-nya), apakah proses lain membutuhkan *critical section* atau tidak.

Awalnya *flag* untuk kedua proses diinisialisai bernilai *false*, yang artinya kedua proses tersebut tidak membutuhkan *critical section*. Jika P<sub>0</sub> ingin mengakses *critical section*, ia akan mengubah *flag*[0] menjadi *true*. Kemudian P<sub>0</sub> akan mengecek apakah P<sub>1</sub> juga



membutuhkan *critical section*, jika *flag[1]* bernilai *false* maka P0 akan menggunakan *critical section*. Namun jika *flag[1]* bernilai *true* maka P0 harus menunggu P1 menggunakan *critical section* dan mengubah *flag[1]* menjadi *false*.

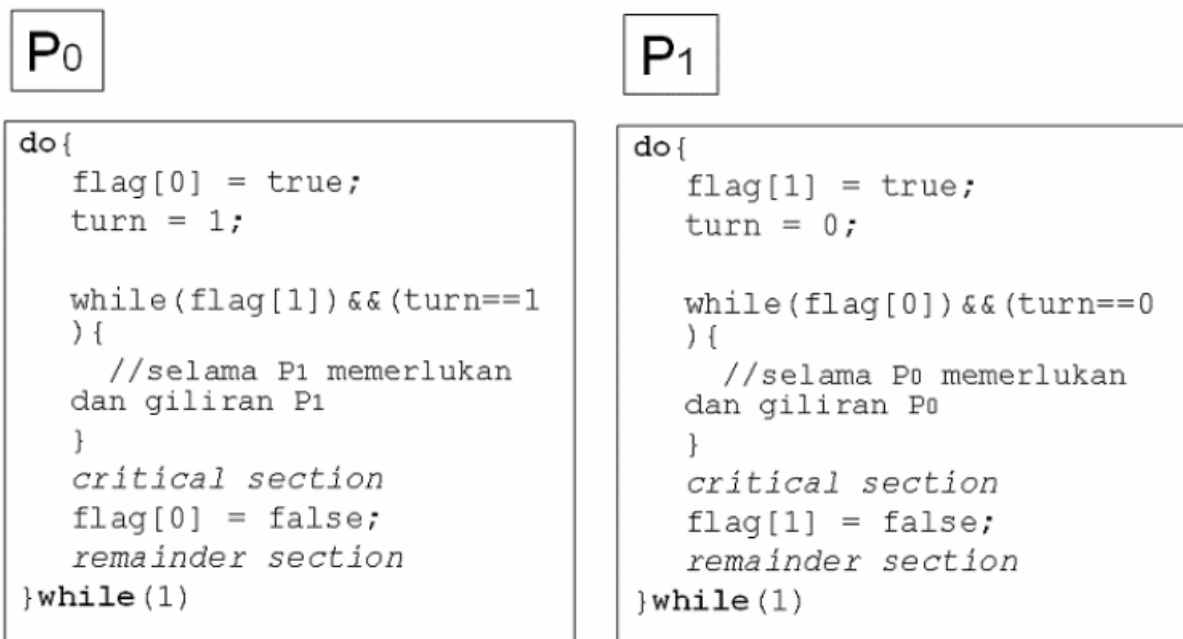
Pada algoritma ini masalah muncul ketika kedua proses secara bersamaan menginginkan *critical section*, kedua proses tersebut akan men- set masing-masing *flag*-nya menjadi *true*. P0 men-set *flag[0] = true*, P1 men-set *flag[1] = true*. Kemudian P0 akan mengecek apakah P1 membutuhkan *critical section*. P0 akan melihat bahwa *flag[1] = true*, maka P0 akan menunggu sampai P1 selesai menggunakan *critical section*. Namun pada saat bersamaan, P1 juga akan mengecek apakah P0 membutuhkan *critical section* atau tidak, ia akan melihat bahwa *flag[0] = true*, maka P1 juga akan menunggu P0 selesai menggunakan *critical section*-nya. Kondisi ini menyebabkan kedua proses yang membutuhkan *critical section* tersebut akan saling menunggu dan "saling mempersilahkan" proses lain untuk mengakses *critical section*, akibatnya malah tidak ada yang mengakses *critical section*. Kondisi ini menunjukkan bahwa Algoritma II tidak memenuhi syarat progress dan syarat bounded waiting, karena kondisi ini akan terus bertahan dan kedua proses harus menunggu selamanya untuk dapat mengakses *critical section*.



# Algoritma III

Algoritma III ditemukan oleh G.L. Petterson pada tahun 1981 dan dikenal juga sebagai Algoritma Petterson. Petterson menemukan cara yang sederhana untuk mengatur proses agar memenuhi *mutual exclusion*. Algoritma ini adalah solusi untuk memecahkan masalah *critical section* pada dua proses.

Ide dari algoritma ini adalah menggabungkan variabel yang di-*sharing* pada Algoritma I dan Algoritma II, yaitu variabel *turn* dan variabel *flag*. Sama seperti pada Algoritma I dan II, variabel *turn* menunjukkan giliran proses mana yang diperbolehkan memasuki *critical section* dan variabel *flag* menunjukkan apakah suatu proses membutuhkan akses ke *critical section* atau tidak.



Gambar 3. Algoritma III

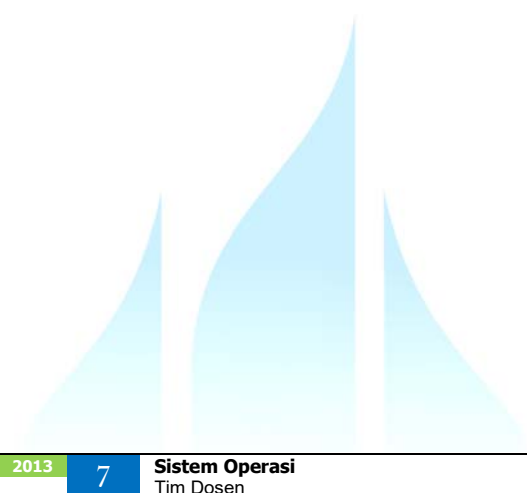
Awalnya *flag* untuk kedua proses diinisialisasi bernilai *false*, yang artinya kedua proses tersebut tidak membutuhkan akses ke *critical section*. Kemudian jika suatu proses ingin memasuki *critical section*, ia akan mengubah *flag*-nya menjadi *true* (memberikan tanda bahwa ia butuh *critical section*) lalu proses tersebut memberikan *turn* kepada lawannya. Jika lawannya tidak menginginkan *critical section* (*flag*-nya *false*), maka proses tersebut dapat menggunakan *critical section*, dan setelah selesai menggunakan *critical section* ia akan mengubah *flag*-nya menjadi *false*. Tetapi apabila proses lawannya juga menginginkan *critical section* maka proses lawan-lah yang dapat memasuki *critical section*, dan proses

tersebut harus menunggu sampai proses lawan menyelesaikan *critical section* dan mengubah *flag*-nya menjadi *false*.

Misalkan ketika P0 membutuhkan *critical section*, maka P0 akan mengubah *flag*[0] = *true*, lalu P0 mengubah *turn* = 1. Jika P1 mempunyai *flag*[1] = *false*, (berapapun nilai *turn*) maka P0 yang dapat mengakses *critical section*. Namun apabila P1 juga membutuhkan *critical section*, karena *flag*[1] = *true* dan *turn* = 1, maka P1 yang dapat memasuki *critical section* dan P0 harus menunggu sampai P1 menyelesaikan *critical section* dan mengubah *flag*[1] = *false*, setelah itu barulah P0 dapat mengakses *critical section*.

Bagaimana bila kedua proses membutuhkan *critical section* secara bersamaan? Proses mana yang dapat mengakses *critical section* terlebih dahulu? Apabila kedua proses (P0 dan P1) datang bersamaan, kedua proses akan menset masing-masing *flag* menjadi *true* (*flag*[0] = *true* dan *flag*[1] = *true*), dalam kondisi ini P0 dapat mengubah *turn* = 1 dan P1 juga dapat mengubah *turn* = 0. Proses yang dapat mengakses *critical section* terlebih dahulu adalah proses yang terlebih dahulu mengubah *turn* menjadi *turn* lawannya. Misalkan P0 terlebih dahulu mengubah *turn* = 1, lalu P1 akan mengubah *turn* = 0, karena *turn* yang terakhir adalah 0 maka P0-lah yang dapat mengakses *critical section* terlebih dahulu dan P1 harus menunggu.

Algoritma III memenuhi ketiga syarat yang dibutuhkan. Syarat *progress* dan *bounded waiting* yang tidak dipenuhi pada Algoritma I dan II dapat dipenuhi oleh algoritma ini karena ketika ada proses yang ingin mengakses *critical section* dan tidak ada yang menggunakan *critical section* maka dapat dipastikan ada proses yang bisa menggunakan *critical section*, dan proses tidak perlu menunggu selamanya untuk dapat masuk ke *critical section*.



# Daftar Pustaka

1. Stalling.W, *Operating System Internals and Design Principle Seventh Edition*, Prentice hall, 2012
2. Silberschatz. A, Galvin. P.B, Gagne. Greg, *Operating System Concepts Ninth perEdition*, Jhon, Wiley & Sons, 2013
3. <http://bebas.vlsm.org/v06/Kuliah/SistemOperasi/BUKU/>



## MODUL PERKULIAHAN

# Sistem Operasi

## Penjadwalan CPU “Uniprocessor”

Fakultas  
Ilmu Komputer

Program Studi  
Teknik Informatika

Tatap Muka

**11**

Kode MK  
87030

Disusun Oleh  
Tim Dosen

### Abstract

Modul ini membahas tentang bagaimana algoritma penjadwalan proses pada CPU

### Kompetensi

Diharapkan mahasiswa mengetahui dan memahami algoritma penjadwalan CPU

# Konsep Dasar

Pada sebuah sistem single-processor, dimana hanya satu proses yang berjalan pada satu waktu. Proses yang lain harus menunggu sampai CPU bebas dan dapat dijadwalkan kembali.

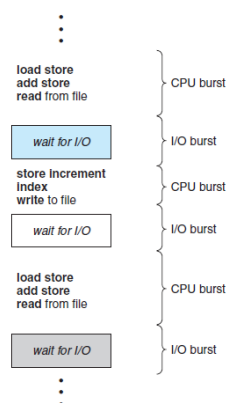
Pada multiprogramming, dimana memiliki beberapa proses yang berjalan pada waktu yang sama, dengan tujuan untuk memaksimalkan penggunaan CPU. Sebuah proses yang akan dieksekusi biasanya harus menunggu untuk menyelesaikan beberapa permintaan I/O.

Pada sebuah sistem komputer sederhana, CPU hanya diam, dan menyebabkan waiting time sehingga mubazir. Dengan multiprogramming, maka penggunaan waktu lebih produktif. Beberapa proses dapat disimpan dalam memori pada satu waktu. Pada saat 1 proses harus menunggu, OS meminta CPU untuk melepaskan proses tersebut dan memberikan CPU ke proses lain. Pola ini terus berlanjut. Setiap waktu satu proses menunggu, proses lain akan mengambil alih penggunaan CPU.

Sebagian besar sumber daya komputer dijadwalkan sebelum penggunaan. CPU merupakan salah satu sumber daya komputer yang utama, sehingga penjadwalan CPU merupakan pusat dari desai OS.

## CPU-I/O Burst Cycle

Sukses nya penjadwalan CPU tergantung pada property proses yang diobservasi, dimana eksekusi proses terdiri dari sebuah cycle eksekusi CPU dan I/O wait. Proses-proses akan berada pada 2 state ini. Proses akan memulai eksekusinya dengan sbuah CPU Burst, yang diikuti I/O burst, kemudian diikuti oleh CPU Burst lain, diikuti I/O Burst lain dan seterusnya. CPU burst terakhir akan menghentikan eksekusi (Gambar 11.1)



Gambar 11.1 CPU-I/O Burst Cycle

# Jenis-jenis penjadwalan CPU

Tujuan dari penjadwalan processor adalah untuk menentukan proses-proses yang akan dieksekusi oleh processor, dengan memperhatikan ukuran kinerja processor seperti response time, throughput, dan efisiensi dari processor.

Jenis-jenis dari aktifitas penjadwalan ini dibagi menjadi 3 yaitu:

1. Long term scheduling

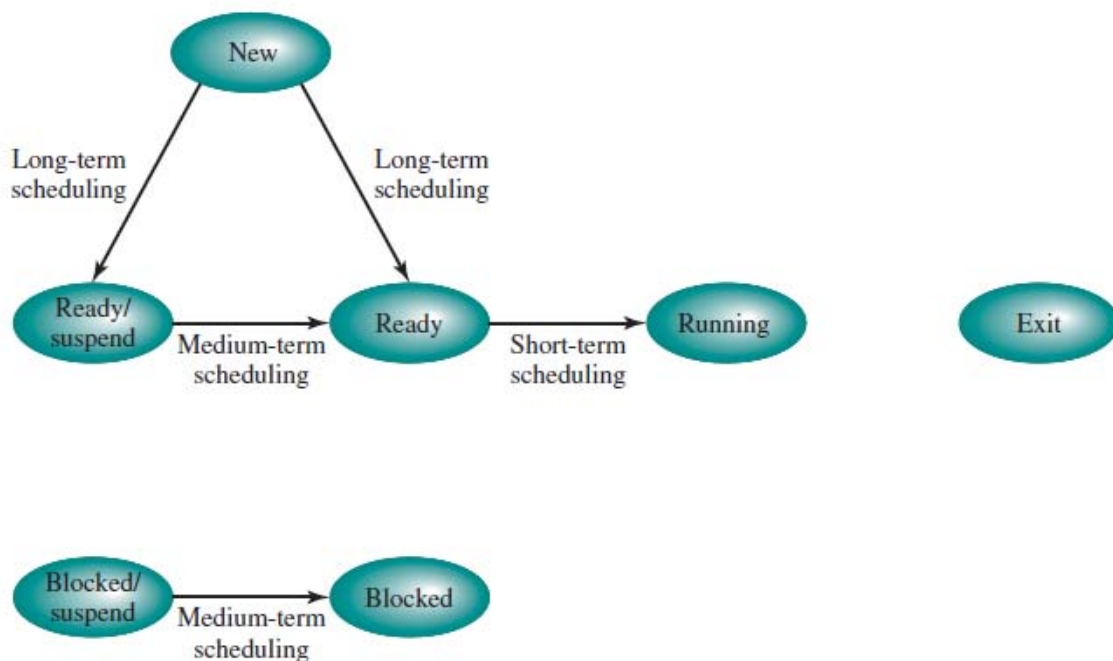
Dilaksanakan pada saat sebuah proses baru diciptakan. Hal ini merupakan sebuah keputusan apakah menambah sebuah proses baru ke sekumpulan proses yang sedang aktif

2. Medium term scheduling

Bagian dari sebuah fungsi swapping. Hal ini merupakan sebuah keputusan apakah menambahkan sebuah proses ke memori dan siap untuk eksekusi.

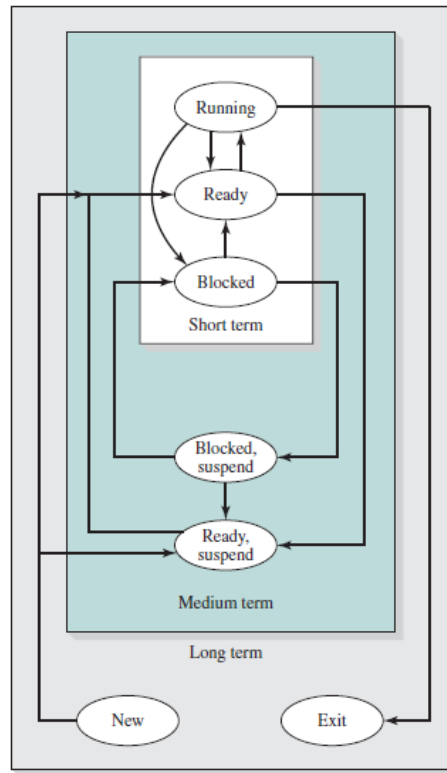
3. Short term scheduling

Keputusan proses yang siap untuk dieksekusi berikutnya.



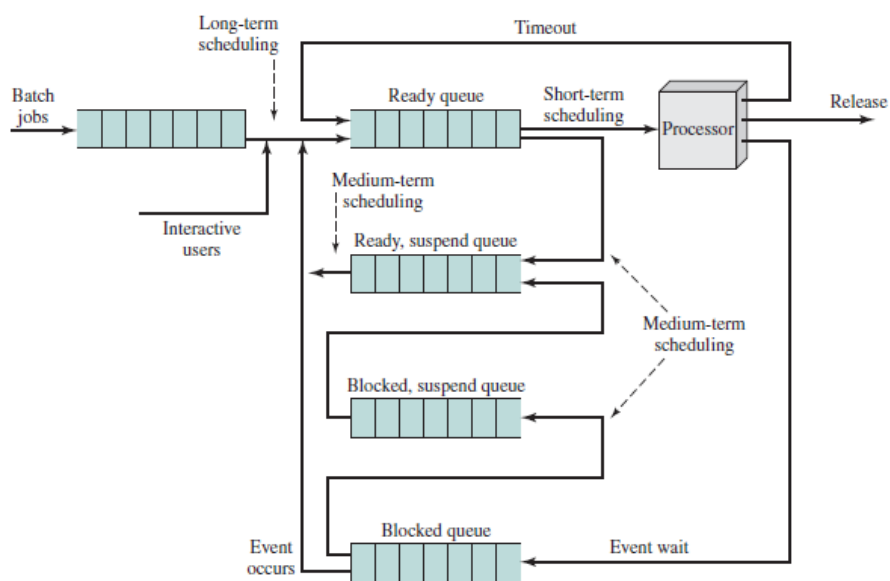
Gambar 11.2 Penjadwalan dan transisi dari state proses

Gambar 11.2 diatas berhubungan dengan fungsi-fungsi penjadwalan untuk diagram transisi dari state poses.



Gambar 11.3 Penjadwalan dan transisi dari state proses

Penjadwalan mempengaruhi kinerja dari sistem karena penjadwalan menentukan proses mana yang akan menunggu dan yang akan dilanjutkan. Hal ini terlihat pada Gambar 11.4, yang menunjukkan antrian yang terlibat dalam transisi state dari sebuah proses. Pada dasarnya penjadwalan terkait dengan pengelolaan antrian untuk meminimalkan delay antrian dan mengoptimasikan kinerja dalam sebuah antrian.



Gambar 11.4 Diagram antrian dari penjadwalan



## Long-Term Scheduling

---

Long-term scheduler menentukan program mana yang akan diberikan ke sistem untuk pemrosesan. Sehingga, long term scheduler melakukan control multiprogramming. Sekali sebuah job atau user program diterima oleh sistem, maka berubah menjadi sebuah proses dan menambahkan ke antrian untuk short term scheduler.

Pada beberapa sistem, sebuah proses baru tercipta mulai pada sebuah kondisi swapped out, dimana pada kasus seperti ini proses tersebut ditambahkan ke medium short scheduler.

Pada sebuah sistem batch, atau sebagian batch pada OS, job yang baru diberikan di teruskan ke disk dan ditangani pada sebuah batch queue. Long term scheduler menciptakan proses proses dari antrian, jika bisa dilakukan. Ada 2 keputusan yang terlibat :

1. Scheduler harus memutuskan kapan OS dapat mengambil satu atau lebih proses tambahan.
2. Scheduler harus memutuskan job mana yang diterima dan dikembalikan ke proses.

Keputusan untuk menciptakan sebuah proses baru umumnya diperoleh dari tingkat multiprogramming yang diinginkan. Semakin banyak proses yang diciptakan, maka semakin kecil persentase waktu untuk setiap proses dieksekusi (banyak proses berkompetisi untuk sejumlah waktu processor). Sehingga, long term scheduler membatasi tingkat multiprogramming untuk menyediakan layanan yang baik untuk sekumpulan proses-proses yang ada. Setiap kali sebuah job terminasi, scheduler akan memutuskan untuk menambahkan satu atau lebih job baru. Sebagai tambahan jika sebagian kecil waktu dari processor kosong melebihi threshold tertentu, long term scheduler akan dipanggil.

Keputusan job mana yang akan diberikan berikutnya, dilakukan dengan cara yang sederhana yaitu First Come First Serve (FCFS). Kriteria yang digunakan adalah prioritas, waktu eksekusi dan kebutuhan I/O.

Sebagai contoh, jika informasi tersedia, scheduler akan mencoba untuk mempertahankan processor bound dan I/O bound proses.

Keputusan dapat tergantung pada sumber daya I/O mana yang diminta, hal ini untuk menjaga keseimbangan penggunaan I/O. Untuk program interaktif pada sebuah sistem time-sharing, sebuah permintaan penciptaan proses dapat dihasilkan oleh tindakan dari user yang mencoba untuk koneksi sistem. User-user time sharing tidak hanya antri dan harus menunggu sampai sistem menerima.

## Medium-Term Scheduling

---

Medium-term scheduling merupakan bagian dari fungsi swapping. Keputusan swapping-in didasarkan pada keharusan untuk mengelola tingkat multiprogramming. Pada sebuah system yang tidak menggunakan virtual memory, manajemen memory merupakan sebuah isu. Sehingga keputusan swapping-in akan mempertimbangkan kebutuhan memori dari proses-proses swapped-out.

## Short-Term Scheduling

---

Pada poin frekuensi eksekusi, long-term scheduler mengeksekusi agak jarang dan membuat keputusan coarse-grained apakah boleh atau tidak mengambil sebuah proses baru dan proses mana yang diambil.

Medium-term scheduler dieksekusi lebih sering untuk membuat sebuah keputusan swapping. Short-term scheduler, dikenal sebagai dispatcher, eksekusi sangat sering dan membuat keputusan fine-grained proses mana yang akan dieksekusi berikutnya.

Short-term scheduler diberikan pada saat sebuah event terjadi yang menyebabkan blocking dari proses saat ini atau menyediakan kesempatan untuk menyela sebuah proses yang berjalan saat ini. Contoh event-event adalah :

- Clock interrupts
- I/O interrupts
- Operating system calls
- Signals (e.g., semaphores)

## CPU Scheduler

---

Pada saat CPU idle, OS harus memilih salah satu dari proses pada ready queue untuk dieksekusi. Pemilihan proses dilakukan oleh **short-term scheduler**, atau CPU scheduler. Scheduler memilih sebuah proses dari proses-proses dalam memory yang siap untuk eksekusi dan mengalokasikan CPU untuk proses tersebut.

## Preemptive Scheduling

---

Penjadwalan CPU diputuskan berdasarkan hal-hal berikut:

1. Ketika sebuah process berpindah dari running state ke waiting state (Contoh, hasil dari sebuah I/O request atau permohonan wait() untuk terminasi child process.
2. Ketika sebuah proses berpindah dari running state ke ready state (Contoh, pada saat interupsi)
3. Ketika proses berpindah dari waiting state ke ready state (sebagai contoh, penyelesaian I/O).
4. Pada saat proses terminasi.

Untuk situasi 1 dan 4, tidak ada pilihan penjadwalan. Sebuah proses baru (Jika satu proses yang ada dalam ready queue) harus dipilih untuk eksekusi. Situasi 2 dan 3 terdapat pilihan. Pada saat penjadwalan berada pada poin 1 dan 4, maka pola penjadwalan disebut **nonpreemptive** atau **cooperative**. Sedangkan yang lain disebut **preemptive**.

Pada penjadwalan nonpreemptive, sekali CPU sudah dialokasikan ke proses, proses mempertahankan CPU sampai proses tersebut melepaskan CPU dengan cara terminasi atau dengan switching ke waiting state. Penjadwalan ini digunakan pada Microsoft Windows 3.x.

Sedangkan penjadwalan preemptive, proses yang berjalan saat ini dapat diinterupsi dan dipindahkan ke Ready state oleh OS. Windows 95 mulai diperkenalkan penjadwalan preemptive dan versi Windows OS berikutnya. Selain Windows, Mac OS X juga menggunakan penjadwalan model preemptive.

## Dispatcher

---

Komponen lain terkait fungsi CPU-scheduling adalah **dispatcher**. Dispatcher adalah module yang memberikan kontrol CPU ke proses yang dipilih oleh short-term scheduler. Fungsi ini melibatkan:

- Switching context
- Switching to user mode
- Jumping to the proper location in the user program to restart that program

Dispatcher harus secepat mungkin, karena meminta setiap process switch. Waktu yang diperlukan dispatcher untuk menghentikan satu proses dan memulai running proses lain disebut sebagai **dispatch latency**.

# Algoritma penjadwalan CPU

## Short-Term Scheduling Criteria

Bagian utama dari short-term scheduling adalah mengalokasikan waktu processor yang bertujuan untuk mengoptimalkan satu atau lebih aspek-aspek perilaku sistem. Secara umum, sekumpulan kriteria

Kriteria yang digunakan untuk evaluasi kebijakan penjadwalan adalah:

1. User-oriented.  
Berhubungan dengan perilaku sistem yang terkait dengan user individu atau proses.
2. Sistem-oriented.  
Focus pada penggunaan processor yang efektif dan efisien.

### User Oriented, Performance Related

**Turnaround time** This is the interval of time between the submission of a process and its completion. Includes actual execution time plus time spent waiting for resources, including the processor. This is an appropriate measure for a batch job.

**Response time** For an interactive process, this is the time from the submission of a request until the response begins to be received. Often a process can begin producing some output to the user while continuing to process the request. Thus, this is a better measure than turnaround time from the user's point of view. The scheduling discipline should attempt to achieve low response time and to maximize the number of interactive users receiving acceptable response time.

**Deadlines** When process completion deadlines can be specified, the scheduling discipline should subordinate other goals to that of maximizing the percentage of deadlines met.

### User Oriented, Other

**Predictability** A given job should run in about the same amount of time and at about the same cost regardless of the load on the system. A wide variation in response time or turnaround time is distracting to users. It may signal a wide swing in system workloads or the need for system tuning to cure instabilities.

### System Oriented, Performance Related

**Throughput** The scheduling policy should attempt to maximize the number of processes completed per unit of time. This is a measure of how much work is being performed. This clearly depends on the average length of a process but is also influenced by the scheduling policy, which may affect utilization.

**Processor utilization** This is the percentage of time that the processor is busy. For an expensive shared system, this is a significant criterion. In single-user systems and in some other systems, such as real-time systems, this criterion is less important than some of the others.

### System Oriented, Other

**Fairness** In the absence of guidance from the user or other system-supplied guidance, processes should be treated the same, and no process should suffer starvation.

**Enforcing priorities** When processes are assigned priorities, the scheduling policy should favor higher-priority processes.

**Balancing resources** The scheduling policy should keep the resources of the system busy. Processes that will underutilize stressed resources should be favored. This criterion also involves medium-term and long-term scheduling.

# Daftar Pustaka

1. Stalling.W, *Operating System Internals and Design Principle Seventh Edition*, Prentice hall, 2012
2. Silberschatz. A, Galvin. P.B, Gagne. Greg, *Operating System Concepts Ninth perEdition*, Jhon, Wiley & Sons, 2013
3. <http://bebas.vlsm.org/v06/Kuliah/SistemOperasi/BUKU/>



## MODUL PERKULIAHAN

# Sistem Operasi

## Penjadwalan CPU “Algoritma”

Fakultas  
Ilmu Komputer

Program Studi  
Teknik Informatika

Tatap Muka

Kode MK  
87030

Disusun Oleh  
Tim Dosen

# 12

### Abstract

Modul ini membahas tentang bagaimana algoritma penjadwalan proses pada CPU

### Kompetensi

Diharapkan mahasiswa mengetahui dan memahami algoritma penjadwalan CPU

# Algoritma Penjadwalan

Penjadwalan CPU harus memberikan solusi dari masalah pengambilan keputusan proses mana dalam Ready Queue yang akan dialokasikan ke CPU.

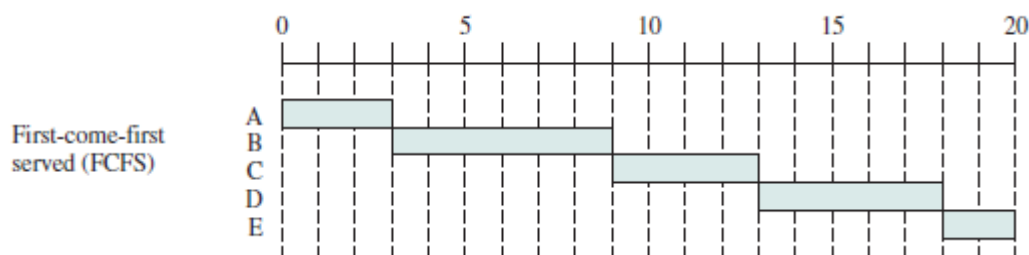
## 1. First Come First Serve

Penjadwalan yang paling sederhana adalah first-come-first served (FCFS), yang dikenal juga dengan nama first-in-first-out (FIFO) atau strict queueing scheme. Dengan pola ini, proses yang meminta CPU pertama dialokasikan juga pertama untuk CPU. Ketika sebuah proses masuk ke ready queue, PCB proses tersebut akan me-link ke ujung antrian. Pada saat CPU bebas, proses tersebut akan dialokasikan ke CPU pada kepala antrian. Proses yang berjalan akan dihapus dari antrian.

### Contoh 1.

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

### Gant Chart



### Hasil

Process	A	B	C	D	E	
Arrival Time	0	2	4	6	8	
Service Time ( $T_s$ )	3	6	4	5	2	Mean
FCFS						
Finish Time	3	9	13	18	20	
Turnaround Time ( $T_T$ )	3	7	9	12	12	8.60
$T_T/T_s$	1.00	1.17	2.25	2.40	6.00	2.56

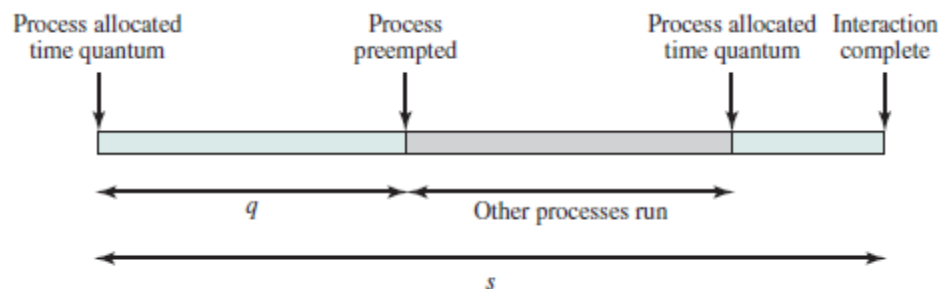
## 2. Round Robin

Algoritma ini sama dengan penjadwalan FCFS, tetapi ditambahkan dengan preemption untuk men-switch antar proses. Sebuah satuan waktu kecil yang disebut time quantum atau time slice ditentukan. Sebuah time quantum, umumnya memiliki panjang antara 10 – 100 milliseconds. Ready queue diperlakukan sebagai sebuah antrian circular.

CPU scheduler mengelilingi ready queue, mengalokasikan CPU ke setiap proses untuk sebuah interval waktu sampai 1 time quantum.

Implementasi Round Robin:

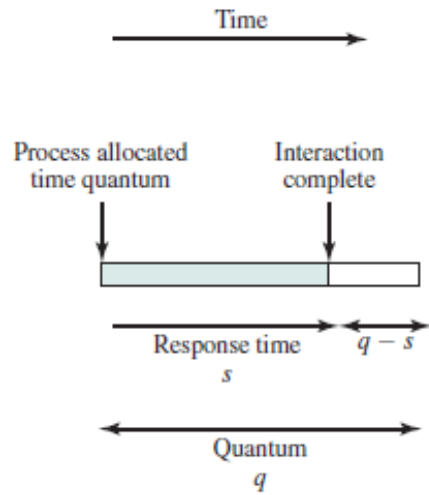
- Ready queue diperlakukan sebagai antrian FIFO dari proses.
- Proses baru ditambahkan ke tail dari ready queue
- CPU scheduler mengambil proses pertama dari ready queue, menyetting timer ke interrupt setelah 1 time quantum, dan melepaskan proses.
  - Sebuah proses bisa memiliki CPU Burst atau service time kurang dari 1 time quantum. Dalam hal ini, proses akan melepaskan CPU dengan sukarela sendiri.



(b) Time quantum less than typical interaction

- Sebuah proses memiliki burst time atau service time lebih besar dari 1 time quantum, timer akan off dan menyebabkan sebuah interupsi ke OS. Context switch akan dieksekusi, dan proses akan ditempatkan ke tail ready queue. Scheduler akan memilih proses berikutnya dalam ready queue.



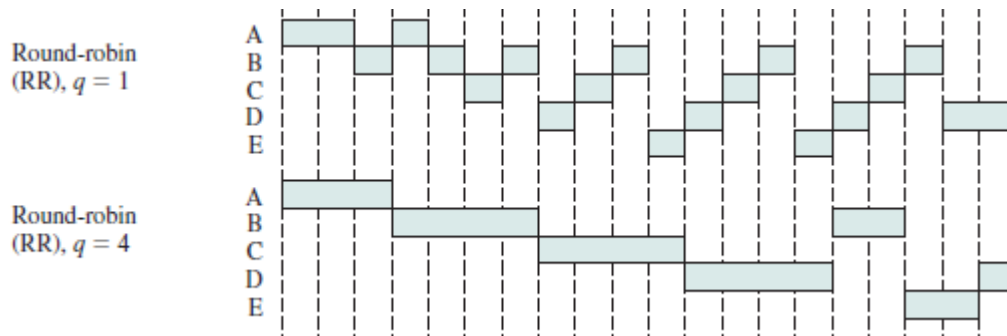


(a) Time quantum greater than typical interaction

Contoh .

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

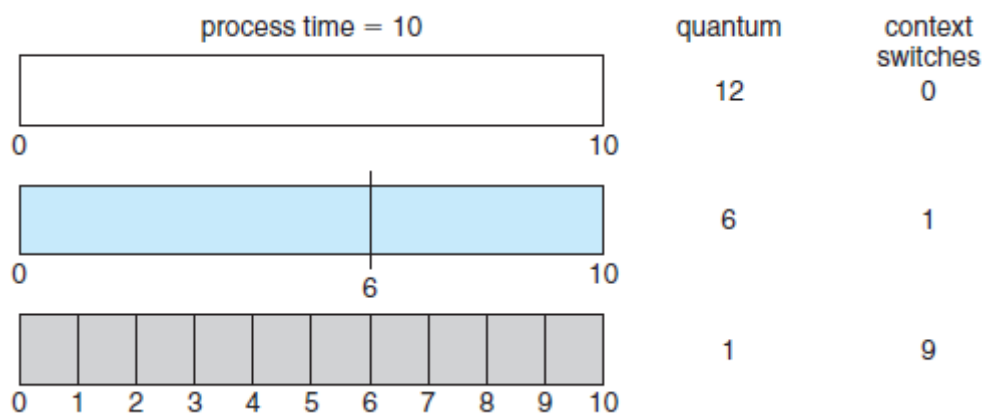
### Gant Chart



## Hasil

Process	A	B	C	D	E	
Arrival Time	0	2	4	6	8	
Service Time ( $T_s$ )	3	6	4	5	2	Mean
<b>RR <math>q = 1</math></b>						
Finish Time	4	18	17	20	15	
Turnaround Time ( $T_T$ )	4	16	13	14	7	10.80
$T_T/T_s$	1.33	2.67	3.25	2.80	3.50	2.71
<b>RR <math>q = 4</math></b>						
Finish Time	3	17	11	20	19	
Turnaround Time ( $T_T$ )	3	15	7	14	11	10.00
$T_T/T_s$	1.00	2.5	1.75	2.80	5.50	2.71

Semakin kecil time quantumnya maka akan meningkatkan jumlah context switch, yang akan menyebabkan semakin lambatnya eksekusi, karena banyak waktu yang diperlukan untuk melakukan context switch.



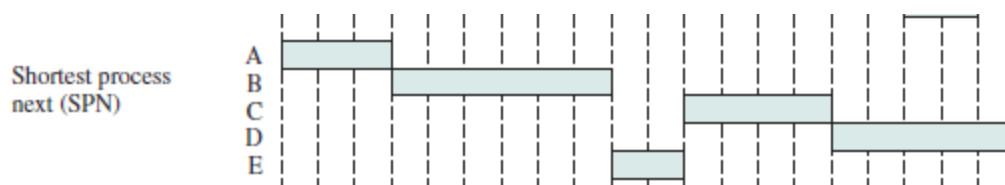
### 3. Shortest Proses Next (SPN)

Algoritma penjadwalan Shortest Process next (SPN) atau shortest-job-first (SJF) berhubungan dengan panjang proses dari CPU Burst atau Service Time dari proses berikutnya. Pada saat CPU tersedia, maka diberikan ke proses yang memiliki service time paling kecil. Jika service time dari 2 proses sama, maka penjadwalan FCFS yang digunakan untuk menentukannya

**Contoh.**

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

**Gant Chart**



**Hasil**

Process	A	B	C	D	E	
Arrival Time	0	2	4	6	8	
Service Time ( $T_s$ )	3	6	4	5	2	Mean
<b>SPN</b>						
Finish Time	3	9	15	20	11	
Turnaround Time ( $T_T$ )	3	7	11	14	3	7.60
$T_T/T_s$	1.00	1.17	2.75	2.80	1.50	1.84

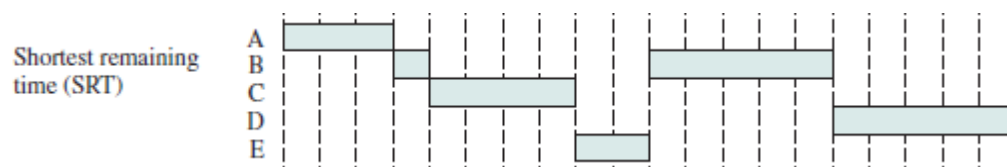
#### 4. Shortest-Remaining-Time-First

Algoritma penjadwalan Shortest Process next (SPN) atau shortest-job-first (SJF) dapat di preemptive atau nonpreemptive. Pemilihan ini muncul pada saat sebuah proses baru datang ke ready queue ketika sebuah proses lama masih dieksekusi. Service time dari proses yang baru datang lebih pendek dibandingkan sisa service time dari proses yang sedang dieksekusi. Algoritma ini akan melakukan preempt proses yang sedang dieksekusi saat ini,

##### Contoh

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

##### Gant Chart



##### Hasil

Process	A	B	C	D	E	
Arrival Time	0	2	4	6	8	
Service Time ( $T_s$ )	3	6	4	5	2	Mean
<b>SRT</b>						
Finish Time	3	15	8	20	10	
Turnaround Time ( $T_T$ )	3	13	4	14	2	7.20
$T_T/T_s$	1.00	2.17	1.00	2.80	1.00	1.59

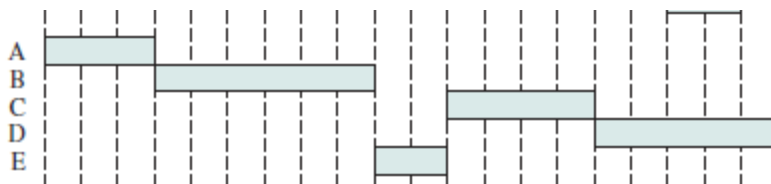
## 5. Priority Scheduling

Sebuah prioritas diberikan ke setiap proses dan CPU dialokasikan ke proses dengan prioritas yang lebih tinggi. Untuk proses dengan prioritas yang sama, akan dijadwalkan dengan menggunakan FCFS.

Contoh adalah algoritma Shortest Process Next (SPN), yang memiliki service time terkecil memiliki prioritas terbesar dan yang memiliki service time terbesar memiliki prioritas lebih kecil.

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

### Gant Chart



### Hasil

Process	A	B	C	D	E	
Arrival Time	0	2	4	6	8	
Service Time ( $T_s$ )	3	6	4	5	2	Mean
Finish Time	3	9	15	20	11	
Turnaround Time ( $T_T$ )	3	7	11	14	3	7.60
$T_T/T_s$	1.00	1.17	2.75	2.80	1.50	1.84

# Daftar Pustaka

1. Stalling.W, *Operating System Internals and Design Principle Seventh Edition*, Prentice hall, 2012
2. Silberschatz. A, Galvin. P.B, Gagne. Greg, *Operating System Concepts Ninth perEdition*, Jhon, Wiley & Sons, 2013
3. <http://bebas.vlsm.org/v06/Kuliah/SistemOperasi/BUKU/>



## MODUL PERKULIAHAN

# Sistem Operasi

## Penjadwalan CPU “Multiprocessor”

Fakultas  
Ilmu Komputer

Program Studi  
Teknik Informatika

Tatap Muka

Kode MK  
87030

Disusun Oleh  
Tim Dosen

# 13

### Abstract

Modul ini membahas tentang bagaimana algoritma penjadwalan proses pada multiprocessor

### Kompetensi

Diharapkan mahasiswa mengetahui dan memahami algoritma penjadwalan multiprocessor

# Penjadwalan Multiprocessor

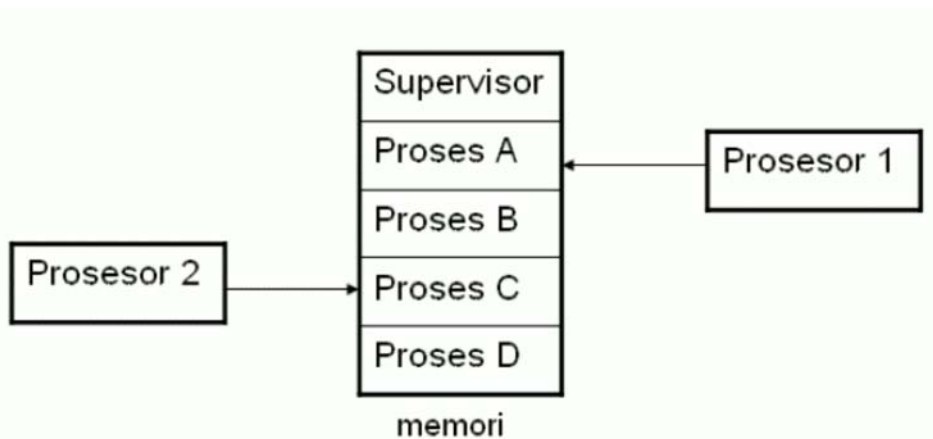
Untuk meningkatkan kinerja, kehandalan, kemampuan komputasi, paralelisme, dan keekonomisan dari suatu sistem, tambahan prosesor dapat diimplementasikan ke dalam sistem tersebut. Sistem seperti ini disebut dengan sistem yang bekerja dengan banyak prosesor (*multiprocessor*). Seperti halnya pada prosesor tunggal, *multiprocessor* juga membutuhkan penjadwalan. Namun pada *multiprocessor*, penjadwalannya jauh lebih kompleks daripada prosesor tunggal karena pada *multiprocessor* memungkinkan adanya *load sharing* antar prosesor yang menyebabkan penjadwalan menjadi lebih kompleks namun kemampuan sistem tersebut menjadi lebih baik. Oleh karena itu, kita perlu mempelajari penjadwalan pada *multiprocessor* berhubung sistem dengan *multiprocessor* akan semakin banyak digunakan karena kemampuannya yang lebih baik dari sistem dengan prosesor tunggal. Ada beberapa jenis dari sistem *multiprocessor*, namun yang akan dibahas dalam bab ini adalah penjadwalan pada sistem *multiprocessor* yang memiliki fungsi- fungsi prosesor yang identik (*homogenous*).

## Penjadwalan Master/Slave

Pendekatan pertama untuk penjadwalan *multiprocessor* adalah penjadwalan *asymmetric multiprocessing* atau bisa disebut juga sebagai penjadwalan *master/slave*. Dimana pada metode ini hanya satu prosesor (*master*) yang menangani semua keputusan penjadwalan, pemrosesan M/K, dan aktivitas sistem lainnya dan prosesor lainnya (*slave*) hanya mengeksekusi proses. Metode ini sederhana karena hanya satu prosesor yang mengakses struktur data sistem dan juga mengurangi data *sharing*.

Dalam teknik penjadwalan *master/slave*, satu prosesor menjaga status dari semua proses dalam sistem dan menjadwalkan kinerja untuk semua prosesor *slave*. Sebagai contoh, prosesor *master* memilih proses yang akan dieksekusi, kemudian mencari prosesor yang *available*, dan memberikan instruksi *Start processor*. Prosesor *slave* memulai eksekusi pada lokasi memori yang dituju. Saat *slave* mengalami sebuah kondisi tertentu seperti meminta M/K, prosesor *slave* memberi interupsi kepada prosesor *master* dan berhenti untuk menunggu perintah selanjutnya. Perlu diketahui bahwa prosesor *slave* yang berbeda dapat ditujukan untuk suatu proses yang sama pada waktu yang berbeda.





**Gambar 13.1. Multiprogramming dengan multiprocessor**

Gambar di atas mengilustrasikan perilaku dari *multiprocessor* yang digunakan untuk *multiprogramming*. Beberapa proses terpisah dialokasikan di dalam memori. Ruang alamat proses terdiri dari halaman-halaman sehingga hanya sebagian saja dari proses tersebut yang berada dalam memori pada satu waktu. Hal ini memungkinkan banyak proses dapat aktif dalam sistem.

### Penjadwalan SMP

Penjadwalan SMP (*Symmetric multiprocessing*) adalah pendekatan kedua untuk penjadwalan multiprocessor. Dimana setiap prosesor menjadwalkan dirinya sendiri (*self scheduling*). Semua proses mungkin berada pada antrian *ready* yang biasa, atau mungkin setiap prosesor memiliki antrian *ready* tersendiri. Bagaimanapun juga, penjadwalan terlaksana dengan menjadwalkan setiap prosesor untuk memeriksa antrian *ready* dan memilih suatu proses untuk dieksekusi. Jika suatu sistem multiprocessor mencoba untuk mengakses dan meng-*update* suatu struktur data, penjadwal dari prosesor-prosesor tersebut harus diprogram dengan hati-hati; kita harus yakin bahwa dua prosesor tidak memilih proses yang sama dan proses tersebut tidak hilang dari antrian. Secara virtual, semua sistem operasi modern mendukung SMP, termasuk Windows XP, Windows 2000, Solaris, Linux, dan Mac OS X.

### Affinity

Data yang paling sering diakses oleh beberapa proses akan memadati *cache* pada prosesor, sehingga akses memori yang sukses biasanya terjadi di memori *cache*. Namun, jika suatu proses berpindah dari satu prosesor ke prosesor lainnya akan mengakibatkan isi dari *cache* memori yang dituju menjadi tidak valid, sedangkan *cache* memori dari prosesor asal harus disusun kembali populasi datanya. Karena mahalnya *invalidating* dan *re-populating* dari *cache*, kebanyakan sistem SMP mencoba untuk mencegah migrasi proses antar prosesor sehingga menjaga proses tersebut untuk berjalan di prosesor yang sama. Hal ini disebut afinitas prosesor (*processor affinity*).

Ada dua jenis afinitas prosesor, yakni:

- *Soft affinity* yang memungkinkan proses berpindah dari satu prosesor ke prosesor yang lain, dan
- *Hard affinity* yang menjamin bahwa suatu proses akan berjalan pada prosesor yang sama dan tidak berpindah. Contoh sistem yang menyediakan *system calls* yang mendukung *hard affinity* adalah Linux.

### Load Balancing

Dalam sistem SMP, sangat penting untuk menjaga keseimbangan *workload* antara semua prosesor untuk memaksimalkan keuntungan memiliki *multiprocessor*. Jika tidak, mungkin satu atau lebih prosesor *idle* disaat prosesor lain harus bekerja keras dengan *workload* yang tinggi. *Load balancing* adalah usaha untuk menjaga *workload* terdistribusi sama rata untuk semua prosesor dalam sistem SMP. Perlu diperhatikan bahwa *load balancing* hanya perlu dilakukan pada sistem dimana setiap prosesor memiliki antrian tersendiri (*private queue*) untuk proses-proses yang berstatus *ready*. Pada sistem dengan antrian yang biasa (*common queue*), *load balancing* tidak diperlukan karena sekali prosesor menjadi *idle*, prosesor tersebut segera mengerjakan proses yang dapat dilaksanakan dari antrian biasa tersebut. Perlu juga diperhatikan bahwa pada sebagian besar sistem operasi kontemporer mendukung SMP, jadi setiap prosesor bisa memiliki *private queue*.

Ada dua jenis *load balancing*, yakni:

- *Push migration*, pada kondisi ini ada suatu *task* spesifik yang secara berkala memeriksa *load* dari tiap-tiap prosesor. Jika terdapat ketidakseimbangan, maka dilakukan perataan dengan memindahkan( *pushing*) proses dari yang kelebihan muatan ke prosesor yang *idle* atau yang memiliki muatan lebih sedikit.
- *Pull migration*, kondisi ini terjadi saat prosesor yang *idle* menarik(*pulling* ) proses yang sedang menunggu dari prosesor yang sibuk.

Kedua pendekatan tersebut tidak harus *mutually exclusive* dan dalam kenyataannya sering diimplementasikan secara paralel pada sistem *load-balancing*.

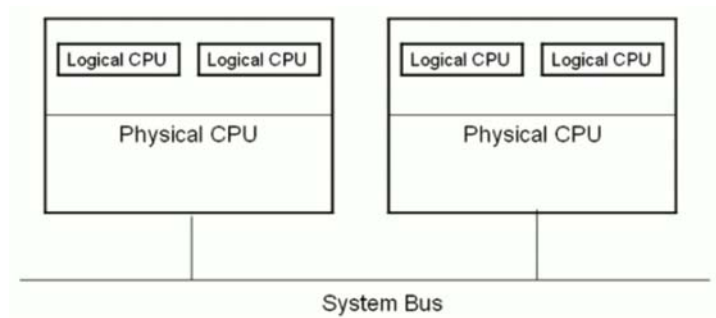
Keuntungan dari *affinity* berlawanan dengan keuntungan dari *load balancing*, yaitu keuntungan menjaga suatu proses berjalan pada satu prosesor yang sama dimana proses dapat memanfaatkan data yang sudah ada pada memori *cache* prosesor tersebut berkebalikan dengan keuntungan menarik atau memindahkan proses dari satu prosesor ke prosesor lain. Dalam kasus *system engineering*, tidak ada aturan tetap keuntungan yang mana yang lebih baik. Walaupun pada beberapa sistem, prosesor *idle* selalu menarik proses dari prosesor *non-idle* sedangkan pada sistem yang lain, proses dipindahkan hanya jika terjadi ketidakseimbangan yang besar antara prosesor.

### ***Symetric Multithreading***

---

Sistem SMP mengizinkan beberapa thread untuk berjalan secara bersamaan dengan menyediakan banyak *physical processor*. Ada sebuah strategi alternatif yang lebih cenderung untuk menyediakan *logical processor* daripada *physical processor* . Strategi ini dikenal sebagai SMT (*Symetric Multithreading*). SMT juga biasa disebut teknologi *hyperthreading* dalam prosesor intel.

Ide dari SMT adalah untuk menciptakan banyak *logical processor* dalam suatu *physical processor* yang sama dan mempresentasikan beberapa prosesor kepada sistem operasi. Setiap *logical processor* mempunyai state arsitekturnya sendiri yang mencakup *general purpose* dan *machine state register*. Lebih jauh lagi, setiap *logical prosesor* bertanggung jawab pada penanganan interupsinya sendiri, yang berarti bahwa interupsi cenderung dikirimkan ke *logical processor* dan ditangani oleh *logical processor* bukan *physical processor*. Dengan kata lain, setiap *logical processor* men- *share resource* dari *physical processor*-nya, seperti *chace* dan bus.



**Gambar 13.2. Symetric Multithreading**

Gambar di atas mengilustrasikan suatu tipe arsitektur SMT dengan dua *physical processor* dengan masing-masing punya dua *logical processor*. Dari sudut pandang sistem operasi, pada sistem ini terdapat empat prosesor.

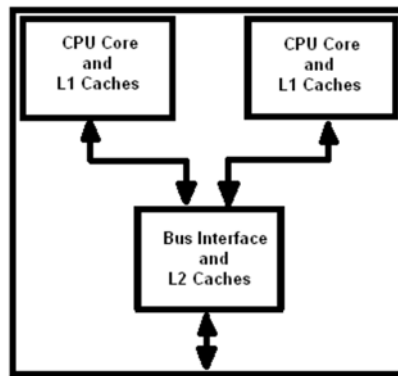
Perlu diketahui bahwa SMT adalah fitur yang disediakan dalam hardware, bukan software, sehingga hardware harus menyediakan representasi state arsitektur dari setiap *logical processor* sebagaimana representasi dari penanganan interupsinya. Sistem operasi tidak perlu didesain khusus jika berjalan pada sistem SMT, akan tetapi performa yang diharapkan tidak selalu terjadi pada sistem operasi yang berjalan pada SMT.

Misalnya, suatu sistem memiliki 2 *physical processor*, keduanya *idle*, penjadwal pertama kali akan lebih memilih untuk membagi thread ke *physical processor* daripada membaginya ke *logical processor* dalam *physical processor* yang sama, sehingga *logical processor* pada satu *physical processor* bisa menjadi sibuk sedangkan *physical processor* yang lain menjadi *idle*.

## **Multicore**

---

*Multicore microprocessor* adalah kombinasi dua atau lebih prosesor independen kedalam sebuah *integrated circuit* (IC). Umumnya, *multicore* mengizinkan perangkat komputasi untuk memeragakan suatu bentuk *thread level paralelism* (TLP) tanpa mengikutsertakan banyak prosesor terpisah. TLP lebih dikenal sebagai *chip-level multiprocessing*.



**Gambar 13.3. Chip CPU *dual-core***

Keuntungan:

- Meningkatkan performa dari operasi *cache snoop (bus snooping)* . *Bus snooping* adalah suatu teknik yang digunakan dalam sistem pembagian memori terdistribusi dan *multiprocessor* yang ditujukan untuk mendapatkan koherensi pada *cache*. Hal ini dikarenakan sinyal antara CPU yang berbeda mengalir pada jarak yang lebih dekat, sehingga kekuatan sinyal hanya berkurang sedikit. Sinyal dengan kualitas baik ini memungkinkan lebih banyak data yang dikirimkan dalam satu periode waktu dan tidak perlu sering di-repeat .
- Secara fisik, desain CPU *multicore* menggunakan ruang yang lebih kecil pada PCB (*Printed Circuit Board*) dibanding dengan desain *multichip* SMP
- Prosesor *dual-core* menggunakan sumber daya lebih kecil dibanding sepasang prosesor *dual-core*
- Desain *multicore* memiliki resiko *design error* yang lebih rendah daripada desain *single-core*

Kerugian:

- Dalam hal sistem operasi, butuh penyesuaian kepada *software* yang ada untuk memaksimalkan kegunaan dari sumberdaya komputasi yang disediakan oleh prosesor *multicore*. Kemampuan prosesor *multicore* untuk meningkatkan performa aplikasi juga bergantung pada penggunaan banyaknya *thread* dalam aplikasi tersebut.
- Dari sudut pandang arsitektur, pemanfaatan daerah permukaan silikon dari desain *single-core* lebih baik daripada desain *multicore*.
- Pengembangan *chip multicore* membuat produksinya menjadi menurun karena semakin sulitnya pengaturan suhu pada chip yang padat.

## Pengaruh *multicore* terhadap *software*

---

Keuntungan *software* dari arsitektur *multicore* adalah kode-kode dapat dieksekusi secara paralel. Dalam sistem operasi, kode-kode tersebut dieksekusi dalam thread-thread atau proses-proses yang terpisah. Setiap aplikasi pada sistem berjalan pada prosesnya sendiri sehingga aplikasi paralel akan mendapatkan keuntungan dari arsitektur *multicore*. Setiap aplikasi harus tertulis secara spesifik untuk memaksimalkan penggunaan dari banyak thread.

Banyak aplikasi *software* tidak dituliskan dengan menggunakan thread-thread yang *concurrent* karena kesulitan dalam pembuatannya. *Concurrency* memegang peranan utama dalam aplikasi paralel yang sebenarnya.

Langkah-langkah dalam mendesain aplikasi paralel adalah sebagai berikut:

- **Partitioning.** Tahap desain ini dimaksudkan untuk membuka peluang awal pengeksekusian secara paralel. Fokus dari tahap ini adalah mempartisi sejumlah besar tugas dalam ukuran kecil dengan tujuan menguraikan suatu masalah menjadi butiran-butiran kecil.
- **Communication.** Tugas-tugas yang telah terpartisi diharapkan dapat langsung dieksekusi secara paralel tapi tidak bisa, karena pada umumnya eksekusi berjalan secara independen. Pelaksanaan komputasi dalam satu tugas membutuhkan asosiasi data antara masing-masing tugas. Data kemudian harus berpindah-pindah antar tugas dalam melangsungkan komputasi. Aliran informasi inilah yang dispesifikasi dalam fase *communication*.
- **Agglomeration.** Pada tahap ini kita pindah dari sesuatu yang abstrak ke yang konkret. Kita tinjau kembali kedua tahap diatas dengan tujuan untuk mendapatkan algoritma pengeksekusian yang lebih efisien. Kita pertimbangkan juga apakah perlu untuk menggumpalkan (*agglomerate*) tugas-tugas pada fase *partition* menjadi lebih sedikit, dengan masing-masing tugas berukuran lebih besar.
- **Mapping.** Dalam tahap yang keempat dan terakhir ini, kita menspesifikasi dimana setiap tugas akan dieksekusi. Masalah *mapping* ini tidak muncul pada *uniprocessor* yang menyediakan penjadwalan tugas.

Pada sisi server, prosesor *multicore* menjadi ideal karena server mengizinkan banyak user untuk melakukan koneksi ke server secara simultan. Oleh karena itu, Web server dan application server mempunyai *throughput* yang lebih baik.

# Daftar Pustaka

1. Stalling.W, *Operating System Internals and Design Principle Seventh Edition*, Prentice hall, 2012
2. Silberschatz. A, Galvin. P.B, Gagne. Greg, *Operating System Concepts Ninth perEdition*, Jhon, Wiley & Sons, 2013
3. <http://bebas.vlsm.org/v06/Kuliah/SistemOperasi/BUKU/>



## MODUL PERKULIAHAN

# Sistem Operasi

## Deadlock

Fakultas  
Ilmu Komputer

Program Studi  
Teknik Informatika

Tatap Muka

**14**

Kode MK  
87030

Disusun Oleh  
Tim Dosen

### Abstract

Modul ini membahas tentang penyebab dan algoritma mengatasi deadlock

### Kompetensi

Diharapkan mahasiswa mengetahui dan memahami penyebab dan algoritma mengatasi deadlock



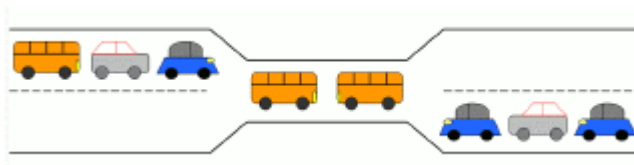
# Pendahuluan

Dalam sistem komputer, terdapat banyak sumber daya yang hanya bisa dimanfaatkan oleh satu proses pada suatu waktu. Contohnya adalah penggunaan sumber daya seperti printer, tape drives dan CD-ROM drives. Dua buah proses yang menggunakan slot yang sama pada tabel proses dapat menyebabkan kerusakan pada sistem. Untuk itu, setiap sistem operasi memiliki mekanisme yang memberikan akses eksklusif pada sumber daya. Pada kenyataannya, proses membutuhkan akses eksklusif untuk beberapa sumber daya sekaligus.

Bayangkan apabila sebuah proses, sebut saja proses A, meminta sumber daya X dan mendapatkannya. Kemudian ada proses B yang meminta sumber daya Y dan mendapatkannya juga. Setelah itu, proses A meminta sumber daya Y dan proses B meminta sumber daya X. Pada situasi tersebut, kedua proses harus ter-block dan menunggu secara terus-menerus. Keadaan seperti itu dinamakan deadlock.

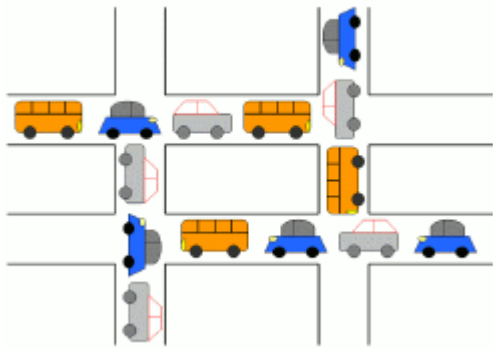
Deadlock secara bahasa berarti buntu atau kebuntuan. Dalam definisi lebih lengkap, deadlock berarti suatu keadaan dimana sistem seperti terhenti dikarenakan setiap proses memiliki sumber daya yang tidak bisa dibagi dan menunggu untuk mendapatkan sumber daya yang sedang dimiliki oleh proses lain. Keadaan seperti ini hanya dapat terjadi pada akses terhadap sumber daya yang tidak bisa dibagi atau non-sharable.

Gambar 14.1. Contoh kasus deadlock pada lalu lintas di jembatan.



Pada contoh di atas, digambarkan ilustrasi dari kejadian deadlock pada dunia nyata, yaitu pada lalu lintas di jembatan. Dapat dilihat bahwa kedua mobil yang berada di tengah-tengah jembatan tidak dapat maju dan hanya menunggu. Penyelesaian dari masalah tersebut adalah salah satu dari mobil tersebut mundur, sehingga mobil yang lain dapat maju. Mobil pada kasus ini adalah proses, sedangkan jembatan adalah sumber daya. Kedua mobil berebut untuk menggunakan sumber daya, namun karena sumber daya tersebut hanya dapat digunakan oleh satu proses saja, maka terjadilah deadlock. Kondisi tersebut bila terjadi dalam waktu yang lama dapat menyebabkan terjadinya starvation.

Gambar 14.2. Contoh kasus deadlock pada lalu lintas di persimpangan



Gambar di atas adalah contoh lain terjadinya deadlock pada dunia nyata. Pada gambar jelas terlihat bahwa lalu lintas terhenti dan terjadi antrian pada empat arah datangnya mobil. Tidak ada mobil yang bisa melanjutkan perjalanan dan hanya menunggu saja. Permasalahan ini dapat dipecahkan dengan cara salah satu dari antrian tersebut mundur dan memberikan kesempatan antrian lain untuk berjalan terlebih dahulu. Kasus seperti ini sangat potensial untuk terjadinya starvation.

Berikut ini diberikan contoh situasi deadlock yang dideskripsikan dengan pseudocode.

### Contoh 1. TestAndSet

```
Mutex M1, M2;

/* Thread 1 */
while (1)
{
    NonCriticalSection()
    Mutex_lock(&M1);
    Mutex_lock(&M2);
    CriticalSection();
    Mutex_unlock(&M2);
    Mutex_unlock(&M1);
}

/* Thread 2 */
while (1)
{
    NonCriticalSection()
    Mutex_lock(&M2);
    Mutex_lock(&M1);
    CriticalSection();
    Mutex_unlock(&M1);
    Mutex_unlock(&M2);
}
```

Misalkan *thread* 1 berjalan dan mengunci M1. Akan tetapi sebelum ia dapat mengunci M2, ia diinterupsi. Kemudian *thread* 2 mulai berjalan dan mengunci M2. Ketika ia mencoba untuk mendapatkan dan mengunci M1, ia terblok karena M1 telah dikunci oleh *thread* 1.

Selanjutnya *thread* 1 berjalan lagi dan mencoba untuk mendapatkan dan mengunci M2, namun terblok karena M2 telah dikunci oleh *thread* 2. Kedua *thread* terblok dan saling menunggu terjadinya sesuatu yang tak pernah akan terjadi.

Kesimpulannya, terjadi *deadlock* yang melibatkan *thread* 1 dan *thread* 2.

## Starvation

---

Pada bagian pendahuluan, telah sama-sama kita ketahui mengenai pengertian dari *deadlock*. Di contoh lalu lintas jembatan, terlihat bahwa kejadian *deadlock* yang berlangsung secara terus-menerus dan tiada akhir dapat menyebabkan terjadinya *starvation*. Akan tetapi, *deadlock* bukanlah satu-satunya penyebab terjadinya *starvation*. Lalu lintas yang didominasi oleh kendaraan-kendaraan dari satu arah pun dapat menyebabkan terjadinya *starvation*.

Akibat yang terjadi adalah kendaraan dari arah lain menjadi terus menunggu giliran untuk berjalan hingga akhirnya mengalami *starvation*.

*Starvation* adalah keadaan dimana satu atau beberapa proses 'kelaparan' karena terus dan terus menunggu kebutuhan sumber dayanya dipenuhi. Namun, karena sumber daya tersebut tidak tersedia atau dialokasikan untuk proses lain, akhirnya proses yang membutuhkan tidak bisa memilikinya.

Kondisi seperti ini merupakan akibat dari keadaan menunggu yang berkepanjangan.

## Model Sistem

---

Keadaan dimana suatu proses yang meminta sumber daya pasti terjadi dalam suatu sistem. Untuk itu dibutuhkan cara pemodelan terhadapnya. Terdapat tipe sumber daya  $R_1$ ,  $R_2$ , ...,  $R_m$ .

Contohnya adalah space pada memori dan juga komponen-komponen M/K. Setiap tipe sumber daya  $R_i$  tersebut memiliki  $W_i$  instances. Misalnya sebuah sumber daya M/K memiliki dua buah instances yang bisa diakses oleh proses.

Sebuah proses dalam melakukan penggunaan terhadap suatu sumber daya melalui langkah-langkah sebagai berikut:

- Request.

Pada langkah ini, pertama kali proses mengajukan diri untuk bisa mendapatkan sumber daya. Proses dapat meminta satu atau lebih sumber daya yang tersedia ataupun yang sedang dimiliki oleh proses yang lain.

- Use.

Selanjutnya, setelah proses mendapatkan sumber daya yang dibutuhkannya, proses akan melakukan eksekusi. Sumber daya digunakan oleh proses sampai proses selesai melakukan eksekusi dan tidak membutuhkan lagi sumber daya tersebut.

- Release

Setelah memanfaatkan sumber daya untuk melakukan eksekusi, proses pun akan melepaskan sumber daya yang dimilikinya. Sumber daya tersebut dibutuhkan oleh proses lain yang mungkin sedang menunggu untuk menggunakan.

## Karakteristik

---

Setelah pada bagian sebelumnya kita telah mengetahui mengenai pengertian dari deadlock dan bagaimana memodelkannya, sekarang kita akan membahas secara mendalam mengenai karakteristik dari terjadinya deadlock.

Karakteristik-karakteristik ini harus dipenuhi keempatnya untuk terjadi deadlock. Namun, perlu diperhatikan bahwa hubungan kausatif antara empat karakteristik ini dengan terjadinya deadlock adalah implikasi.

Deadlock mungkin terjadi apabila keempat karakteristik terpenuhi. Empat kondisi tersebut adalah:

1. Mutual Exclusion.

Kondisi yang pertama adalah mutual exclusion yaitu proses memiliki hak milik pribadi terhadap sumber daya yang sedang digunakannya. Jadi, hanya ada satu proses yang menggunakan suatu sumber daya. Proses lain yang juga ingin menggunakannya harus menunggu hingga sumber daya tersebut dilepaskan oleh proses yang telah selesai menggunakannya. Suatu proses hanya dapat menggunakan secara langsung sumber daya yang tersedia secara bebas.

2. Hold and Wait.

Kondisi yang kedua adalah hold and wait yaitu beberapa proses saling menunggu sambil menahan sumber daya yang dimilikinya. Suatu proses yang memiliki minimal satu buah sumber daya melakukan request lagi terhadap sumber daya. Akan tetapi, sumber daya yang dimintanya sedang dimiliki oleh proses yang lain. Pada saat yang sama, kemungkinan adanya proses lain yang juga mengalami hal serupa dengan proses pertama cukup besar terjadi.

Akibatnya, proses-proses tersebut hanya bisa saling menunggu sampai sumber daya yang dimintanya dilepaskan. Sambil menunggu, sumber daya yang telah dimilikinya pun tidak akan dilepas. Semua proses itu pada akhirnya saling menunggu dan menahan sumber daya miliknya.

3. No Preemption.

Kondisi yang selanjutnya adalah no preemption yaitu sebuah sumber daya hanya dapat dilepaskan oleh proses yang memilikinya secara sukarela setelah ia selesai menggunakannya.

Proses yang menginginkan sumber daya tersebut harus menunggu sampai sumber daya tersedia, tanpa bisa merebutnya dari proses yang memilikinya.

4. Circular Wait.

Kondisi yang terakhir adalah circular wait yaitu kondisi membentuk siklus yang berisi proses-proses yang saling membutuhkan. Proses pertama membutuhkan sumber daya yang dimiliki proses kedua, proses kedua membutuhkan sumber daya milik proses ketiga, dan seterusnya sampai proses ke  $n-1$  yang membutuhkan sumber daya milik proses ke  $n$ . Terakhir, proses ke  $n$  membutuhkan sumber daya milik proses yang pertama. Yang terjadi adalah proses-proses tersebut akan selamanya menunggu. Circular wait oleh penulis diistilahkan sebagai 'Lingkaran Setan' tanpa ujung.

## Penanganan

---

Secara umum terdapat 4 cara untuk menangani keadaan deadlock, yaitu:

1. Pengabaian.

Maksud dari pengabaian di sini adalah sistem mengabaikan terjadinya deadlock dan pura-pura tidak tahu kalau deadlock terjadi. Dalam penanganan dengan cara ini dikenal istilah ostrich algorithm. Pelaksanaan algoritma ini adalah sistem tidak mendeteksi adanya deadlock dan secara otomatis mematikan proses atau program yang mengalami deadlock.

Kebanyakan sistem operasi yang ada mengadaptasi cara ini untuk menangani keadaan deadlock. Cara penanganan dengan mengabaikan deadlock banyak dipilih karena kasus deadlock tersebut jarang terjadi dan relatif rumit dan kompleks untuk diselesaikan. Sehingga biasanya hanya diabaikan oleh sistem untuk kemudian diselesaikan masalahnya oleh user dengan cara melakukan terminasi dengan Ctrl+Alt+Del atau melakukan restart terhadap komputer.

## 2. Pencegahan.

Penanganan ini dengan cara mencegah terjadinya salah satu karakteristik deadlock. Penanganan ini dilaksanakan pada saat deadlock belum terjadi pada sistem. Intinya memastikan agar sistem tidak akan pernah berada pada kondisi deadlock. Akan dibahas secara lebih mendalam pada bagian selanjutnya.

## 3. Penghindaran.

Menghindari keadaan deadlock. Bagian yang perlu diperhatikan oleh pembaca adalah bahwa antara pencegahan dan penghindaran adalah dua hal yang berbeda.

Pencegahan lebih kepada mencegah salah satu dari empat karakteristik deadlock terjadi, sehingga deadlockpun tidak terjadi. Sedangkan penghindaran adalah memprediksi apakah tindakan yang diambil sistem, dalam kaitannya dengan permintaan proses akan sumber daya, dapat mengakibatkan terjadi deadlock. Akan dibahas secara lebih mendalam pada bagian selanjutnya.

## 4. Pendeteksian dan Pemulihan.

Pada sistem yang sedang berada pada kondisi deadlock, tindakan yang harus diambil adalah tindakan yang bersifat represif. Tindakan tersebut adalah dengan mendeteksi adanya deadlock, kemudian memulihkan kembali sistem. Proses pendeteksian akan menghasilkan informasi apakah sistem sedang deadlock atau tidak serta proses mana yang mengalami deadlock. Akan dibahas secara lebih mendalam pada bagian selanjutnya.

## Pencegahan

---

Pencegahan deadlock dapat dilakukan dengan cara mencegah salah satu dari empat karakteristik terjadinya deadlock.

Berikut ini akan dibahas satu per satu cara pencegahan terhadap empat karakteristik tersebut.

1. Mutual Exclusion.

Kondisi mutual exclusion pada sumber daya adalah sesuatu yang wajar terjadi, yaitu pada sumber daya yang tidak dapat dibagi (non-sharable). Sedangkan pada sumber daya yang bisa dibagi tidak ada istilah mutual exclusive. Jadi, pencegahan kondisi yang pertama ini sulit karena memang sifat dasar dari sumber daya yang tidak dapat dibagi.

2. Hold and Wait.

Untuk kondisi yang kedua, sistem perlu memastikan bahwa setiap kali proses meminta sumber daya, ia tidak sedang memiliki sumber daya lain. Atau bisa dengan proses meminta dan mendapatkan sumber daya yang dimilikinya sebelum melakukan eksekusi, sehingga tidak perlu menunggu.

3. No Preemption.

Pencegahan kondisi ini dengan cara membolehkan terjadinya preemption. Maksudnya bila ada proses yang sedang memiliki sumber daya dan ingin mendapatkan sumber daya tambahan, namun tidak bisa langsung dialokasikan, maka akan preempted. Sumber daya yang dimiliki proses tadi akan diberikan pada proses lain yang membutuhkan dan sedang menunggu. Proses akan mengulang kembali eksekusinya setelah mendapatkan semua sumber daya yang dibutuhkannya, termasuk sumber daya yang dimintanya terakhir.

#### 4. Circular Wait.

Kondisi 'lingkaran setan' ini dapat 'diputus' dengan jalan menentukan total kebutuhan terhadap semua tipe sumber daya yang ada. Selain itu, digunakan pula mekanisme enumerasi terhadap tipe-tipe sumber daya yang ada. Setiap proses yang akan meminta sumber daya harus meminta sumber daya dengan urutan yang menaik. Misalkan sumber daya printer memiliki nomor 1 sedangkan CD-ROM memiliki nomor 3. Proses boleh melakukan permintaan terhadap printer dan kemudian CD-ROM, namun tidak boleh sebaliknya.



# Daftar Pustaka

1. Stalling.W, *Operating System Internals and Design Principle Seventh Edition*, Prentice hall, 2012
2. Silberschatz. A, Galvin. P.B, Gagne. Greg, *Operating System Concepts Ninth perEdition*, Jhon, Wiley & Sons, 2013
3. <http://bebas.vlsm.org/v06/Kuliah/SistemOperasi/BUKU/>