Bab 3

Proses Proses

POKOK BAHASAN:

- ✓ Konsep Proses
- ✓ Penjadwalan Proses
- ✓ Operasi pada Proses
- ✓ Kerjasama antar Proses
- ✓ Komunikasi antar Proses
- ✓ Thread

TUJUAN BELAJAR:

Setelah mempelajari materi dalam bab ini, mahasiswa diharapkan mampu:

- ✓ Memahami tentang konsep pada proses dan penjadwalan proses
- ✓ Memahami operasi pembuatan dan penghapusan proses
- ✓ Memahami kerjasana dan komunikasi antar proses.
- ✓ Memahami konsep multi thread, model multi thread dan contoh implementasi thread

3.1 KONSEP PROSES

Sistem operasi mengeksekusi berbagai jenis program. Pada sistem batch program tersebut biasanya disebut dengan *job*, sedangkan pada sistem time sharing, program disebut dengan program user atau *task*. Beberapa buku teks menggunakan istilah *job* atau *proses*.

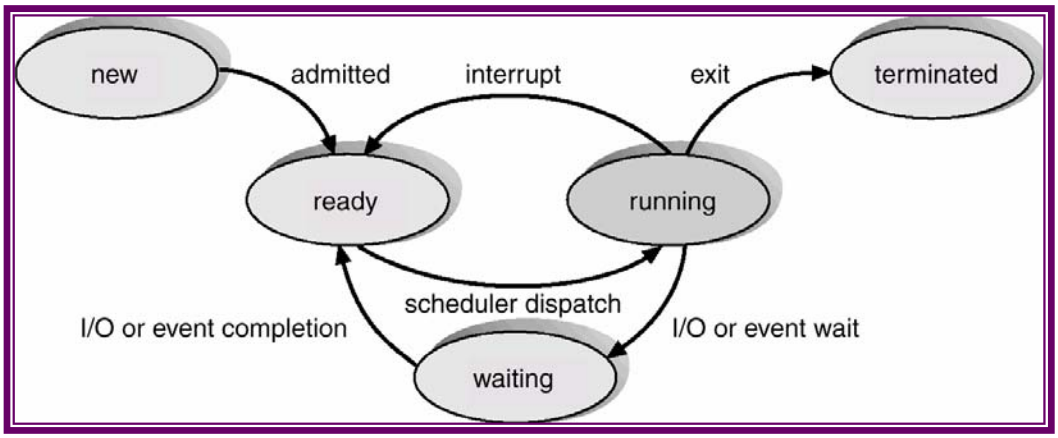
Proses adalah program yang sedang dieksekusi. Eksekusi proses dilakukan secara berurutan. Dalam suatu proses terdapat *program counter, stack* dan daerah data.

3.1.1 Status Proses

Meskipun tiap-tiap proses terdiri dari suatu kesatuan yang terpisah namun adakalanya proses-proses tersebut butuh untuk saling berinteraksi. Satu proses bisa dibangkitkan dari output proses lainnya sebagai input.

Pada saat proses dieksekusi, akan terjadi perubahan status. Status proses didefiniskan sebagai bagian dari aktivitas proses yang sedang berlangsung saat itu. Gambar 3-1 menunjukkan diagram status proses. Status proses terdiri dari :

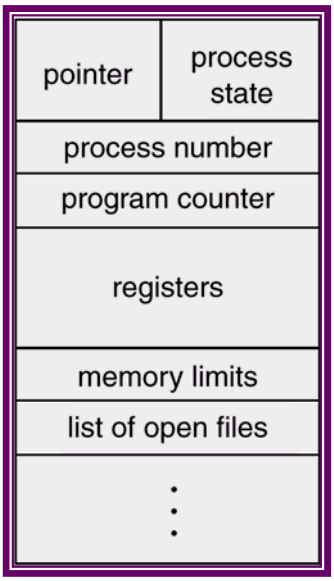
a. *New*: proses sedang dibuat.
b. *Running*: proses sedang dieksekusi.
c. *Waiting*: proses sedang menunggu beberapa *event* yang akan terjadi (seperti menunggu untuk menyelesaikan I/O atau menerima sinyal).
d. *Ready*: proses menunggu jatah waktu dari CPU untuk diproses.
e. *Terminated*: proses telah selesai dieksekusi.



Gambar 3-1: *Perubahan status proses*

3.1.2 Process Control Block (PCB)

Masing-masing proses direpresentasikan oleh Sistem Operasi dengan menggunakan *Process Control Block* (PCB), seperti yang terlihat pada Gambar 3-2.



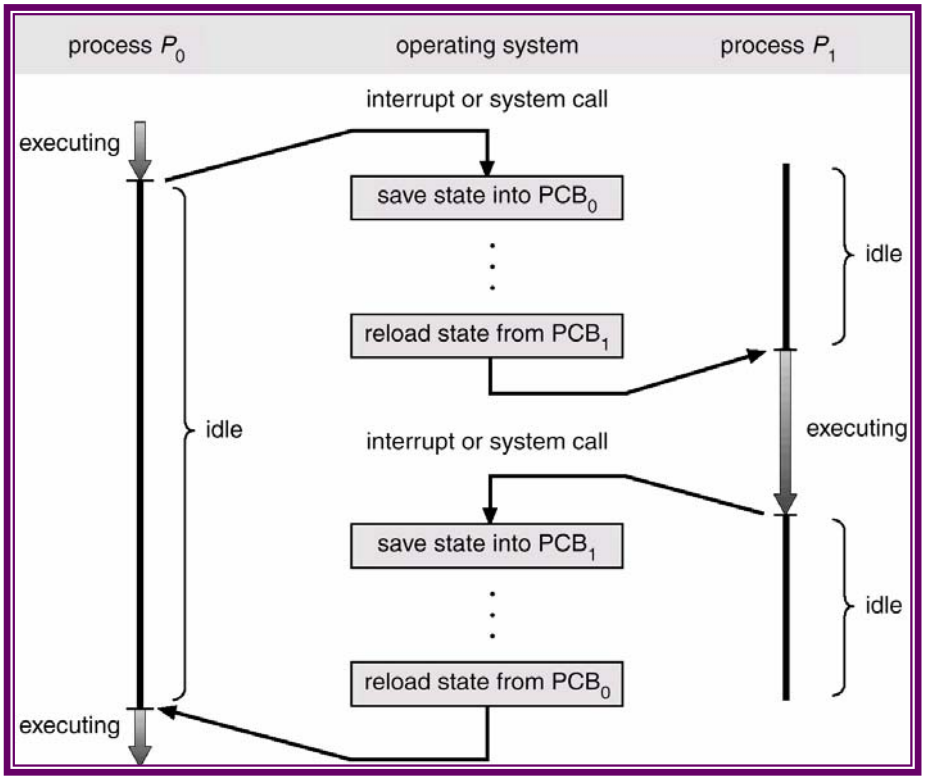
Gambar 3-2: *Process Control Block*

Informasi yang terdapat pada setiap proses meliputi :

a. Status Proses. *New, ready, running, waiting* dan *terminated*.
b. *Program Counter*. Menunjukkan alamat berikutnya yang akan dieksekusi oleh proses tersebut.
c. *CPU Registers*. *Register* bervariasi tipe dan jumlahnya tergantung arsitektur komputer yang bersangkutan. *Register-register* tersebut terdiri-atas: *accumulator, index register, stack pointer*, dan *register* serbaguna dan beberapa informasi tentang kode kondisi.

   Selama *Program Counter* berjalan, status informasi harus disimpan pada saat terjadi interrupt. Gambar 3-3 menunjukkan *switching* proses dari satu proses ke proses berikutnya.
d. Informasi Penjadwalan CPU. Informasi tersebut berisi prioritas dari suatu proses, pointer ke antrian penjadwalan, dan beberapa parameter penjadwalan yang lainnya.
e. Informasi Manajemen Memori. Informasi tersebut berisi nilai (basis) dan limit register, page table, atau segment table tergantung pada sistem memory yang digunakan oleh SO.
f. Informasi Accounting. Informasi tersebut berisi jumlah CPU dan real time yang digunakan, time limits, account numbers, jumlah job atau proses, dll.

g. Informasi Status I/O. Informasi tersebut berisi deretan I/O device (seperti tape driver) yang dialokasikan untuk proses tersebut, deretan file yang dibuka, dll.



Gambar 3-3: *Perpindahan CPU dari satu proses ke proses lain*

3.2 PENJADWALAN PROSES

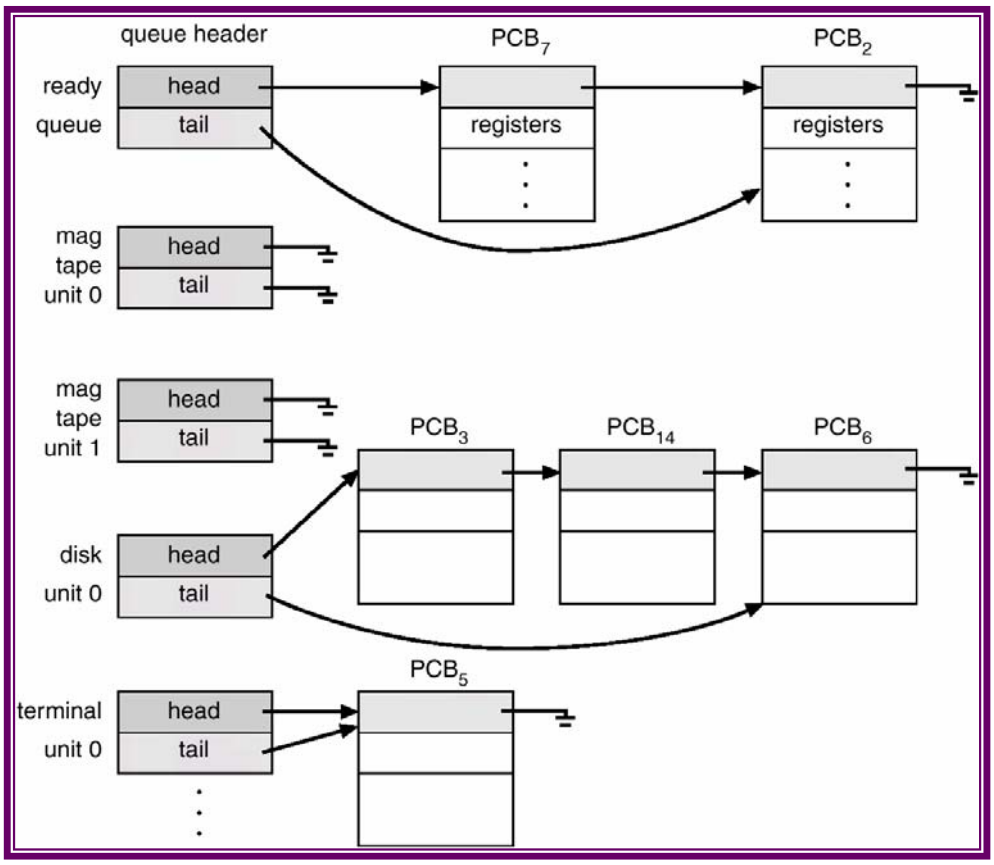
3.2.1 Antrian Penjadwalan

Penjadwalan direpresentasikan dalam bentuk antrian yang disimpan sebagai linkedlist dan berisi pointer awal dan akhir PCB. Tiap-tiap PCB memiliki suatu pointer field yang menunjuk ke proses berikutnya. Jenis-jenis antrian penjadwalan adalah sebagai berikut :

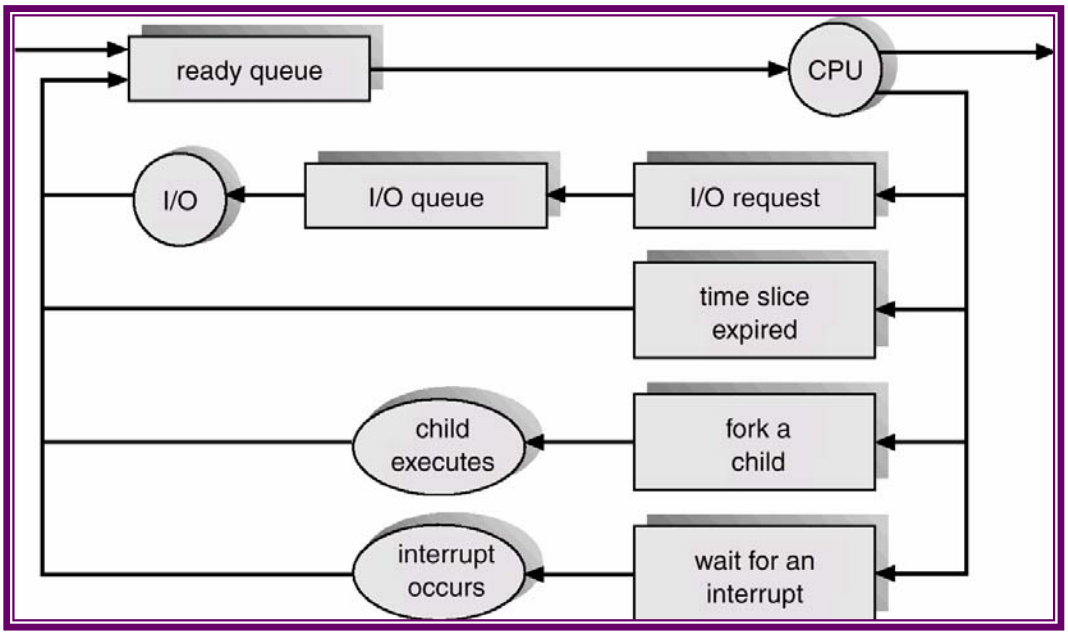
- *Job Queue*. Semua proses yang masuk pada suatu sistem akan diletakkan ke dalam *job queue*.

- *Ready Queue*. Sedangkan proses-proses yang ada di memori utama dan menunggu untuk dieksekusi diletakkan pada suatu list yang disebut dengan *ready queue*. Pada antrian ini berisi
- *Device Queue*. Deretan proses yang sedang menunggu peralatan I/O tertentu disebut dengan *device queue*.

Setiap proses dapat berpindah dari satu antrian ke antrian lain. Gambar 3-4 menunjukkan contoh *ready queue* dan *device queue*. Representasi dari penjadwalan proses dapat dilihat pada Gambar 3-5.



Gambar 3-4: *Ready queue dan device queue*



Gambar 3-4: *representasi penjadwalan proses*

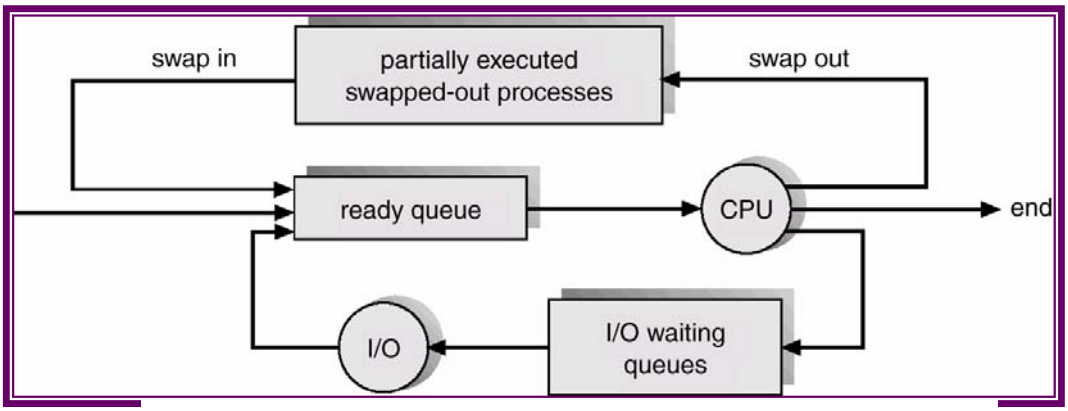
3.2.2 Penjadwal (*Scheduler*)

Terdapat dua bentuk penjadwal, yaitu:

a. *Longterm-Scheduler* (*job scheduler*), menyeleksi proses-proses mana yang harus dibawa ke *ready queue.*

b. *Short-term Scheduler* (*CPU scheduler*), memilih proses-proses yang siap untuk dieksekusi, dan mengakolakasikan CPU ke salah satu dari proses-proses tersebut.

Selain kedua jenis penjadwal diatas terdapat satu jenis penjadwal yang disebut dengan *medium-term scheduler*. Gambar 3-5 merupakan *medium-term scheduler*.



Gambar 3-5: *medium-term scheduler*

Short-term scheduler terjadi sangat sering (dalam milidetik), jadi setiap proses dijadwal dengan cepat, sedangkan *long-term scheduler* terjadi sangat jarang (dalam detik atau menit), sehingga setiap proses dijadwal dengan lambat. *Long-term scheduler* digunakan untuk mengontrol tingkat multiprogramming.

Secara umum, proses dapat digambarkan sebagai :

- *I/O bound process*, yaitu proses-proses yang membutuhkan lebih banyak waktu untuk menjalankan I/O daripada melakukan komputasi, sehingga CPU burst yang dibutuhkan lebih singkat.
- *CPU bound process*, yaitu proses-proses yang membutuhkan lebih banyak waktu untuk melakukan komputasi daripada menjalankan I/O sehingga CPU burst yang dibutuhkan lebih lama.

3.2.3 *Context Switch*

Ketika CPU berpindah dari proses satu ke proses lainnya, sistem harus menyimpan status dari proses yang lama dan membuka state proses baru yang sudah disimpan. *Contex switch* adalah proses penyimpanan status proses dan mengambil status proses yang baru pada saat terjadi switching. Pada saat terjadi perpindahan proses, sistem tidak bekerja. Waktu *context switch* tergantung pada perangkat keras yang digunakan.

3.3 OPERASI PADA PROSES

Terdapat dua operasi pada proses, yaitu pembuatan proses (*process creation*) dan penghentian proses (*process deletion*).

3.3.1 Pembuatan Proses

Ada beberapa aktifitas berkenaan dengan pembuatan proses, antara lain :

a. Memberi identitas (nama) pada proses yang dibuat;
b. Menyisipkan proses pada list proses atau tabel proses;
c. Menentukan prioritas awal proses;

d. Membuat PCB;
e. Mengalokasikan resource awal bagi proses tersebut.

Ada beberapa kejadian yang menyebabkan pembuatan suatu proses baru, antara lain:

a. Pada lingkungan batch sebagai tambahan atas pemberian job. Setelah menciptakan proses baru, sistem operasi melanjutkan untuk membaca job selanjutnya.
b. Pada ingkungan interaktif, pada saat user baru saja logam;
c. Sebagai tanggapan atas suatu aplikasi (seperti: mencetak file, sistem operasi dapat menciptakan proses yang akan mengelola pencetakan itu);
d. Proses menciptakan proses lain (child).

Selama eksekusi, suatu proses mungkin akan membuat suatu proses yang baru. Proses tersebut dinamakan parent, sedangkan proses yang dibuat dinamakan child. Proses pembuatan proses anak membentuk pohon proses.

Pembagian sumber daya :

- Parent dan child membagi semua sumber daya yang ada
- Child menggunakan sebagian dari sumber daya yang digunakan parent
- Parent dan child tidak membagi sumber daya

Bentuk eksekusi :

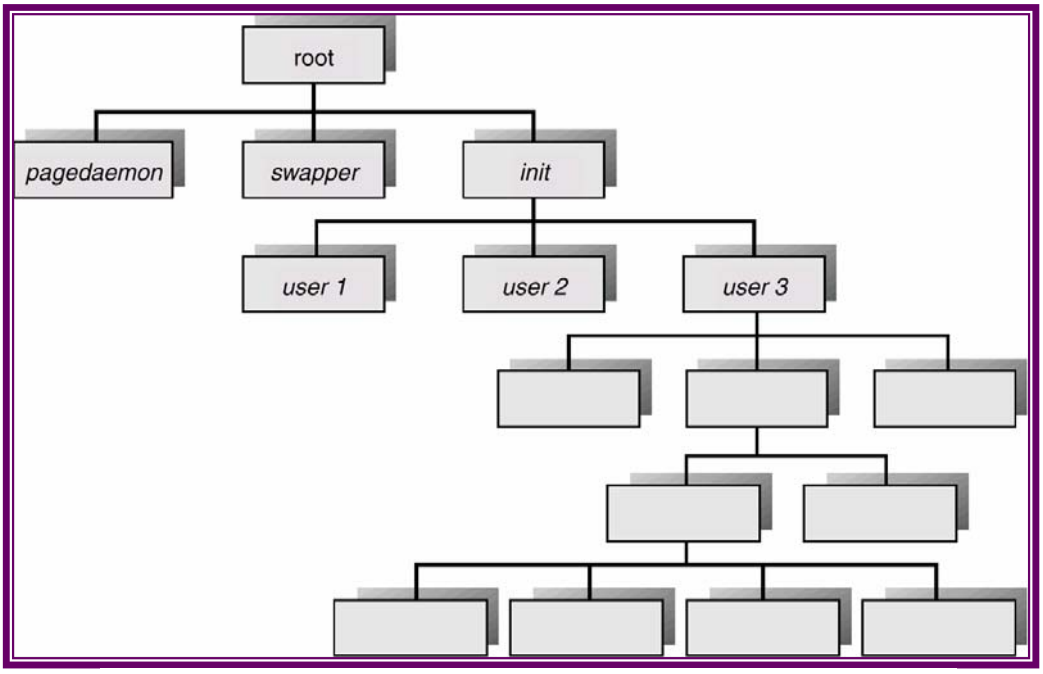
- Parent melanjutkan eksekusi beriringan dengan children.
- Parent menunggu hingga beberapa atau seluruh children selesai.

Bentuk ruang alamat :

- Child adalah duplikat dari proses parent.
- Child mempunyai program yang diambil dari dirinya.

Pada UNIX, parent akan membentuk child dengan menggunakan system call **fork**. Setelah pemanggilan **fork**, parent kembali berjalan secara pararel dengan child. Demikian pula, child dapat memanggil **fork** untuk membentuk child lainnya. Sistem call **exec** digunakan setelah system call **fork** mengganti alamat memori proses dengan program baru. Lain halnya dengan DOS, pada MS-DOS, system call akan memanggil binary file tertentu yang ada pada memori dan mengeksekusinya sebagai child. Parent akan running kembali setelah child selesai eksekusi. Dengan demikian parent dan child

tidak dapat berjalan secara pararel. Bentuk pohon proses pada UNIX dapat dilihat pada Gambar 3-6.



Gambar 3-6: *Bentuk pohon proses pada UNIX*

3.3.2 Penghentian Proses

Suatu proses berhenti jika telah menyelesaikan pernyataan terakhir, dan meminta pada sistem operasi untuk menghapusnya dengan menggunakan system call **exit**. Proses mengembalikan semua data (output) ke parent proses melalui system call **wait**. Kemudian proses dihapus dari list atau tabel sistem, dilanjutkan dengan menghapus PCB.

Penghapusan proses ini akan menjadi sangat kompleks jika ternyata proses yang akan dihentikan tersebut membuat proses-proses yang lain. Pada beberapa sistem, proses-proses anak akan dihentikan secara otomatis jika proses induknya berhenti. Namun, ada beberapa sistem yang menganggap bahwa proses anak ini terpisah dengan induknya, sehingga proses anak tidak ikut dihentikan secara otomatis pada saat proses induk dihentikan.

Parent dapat menghentikan eksekusi proses child dengan menggunakan system call **abort**. Proses anak dihentikan parent karena beberapa alasan, antara lain :

- Child mengalokasikan sumber daya melampaui batas
- Tugas child tidak dibutuhkan lebih lanjut
- Parent berhenti, karena system operasi tidak mengijinkan child untuk melanjutkan jika parent berhenti dan terminasi dilanjutkan.

3.4 PROSES YANG SALING BEKERJA SAMA (COOPERATING PROCESS)

Proses-proses yang dieksekusi oleh sistem operasi mungkin berupa proses-proses yang terpisah (*independence*) atau proses-proses yang saling bekerja sama (*cooperate*). Proses yang terpisah adalah proses yang tidak berakibat atau diakibatkan oleh eksekusi dari proses lain. Sedangkan proses yang saling bekerja sama adalah proses yang dapat berakibat atau diakibatkan oleh eksekusi dari proses lain. Contoh :

P_0 menunggu printer

P_1 menunggu disk drive

Apabila proses terpisah, meskipun P_1 ada dibelakang P_0, namun jika disk drive nganggur, P_1 bisa dieksekusi terlebih dahulu. Sebaliknya jika proses tersebut saling bekerjasama maka eksekusi pada suatu proses akan sangat berpengaruh pada proses yang lain, karena mereka saling berbagi data. Contoh :

P_1: ..., ..., ..., ..., P_2, ...

P_2: ..., ..., ..., ..., ..., ...

Misalkan P_1 adalah program dalam *MS-Word* dan P_2 adalah program *Paintbrush*. *MS-Word* memanggil *Paintbrus*h. Hal ini akan membutuhkan waktu penyimpanan yang cukup besar. Sehingga perlu *swapping* (memindahkan data yang tidak segera dipakai dari memori utama ke memori sekunder.

Keuntungan proses yang saling bekerja sama adalah terjadi pembagian informasi, meningkatkan kecepatan komputasi, proses dapat dibagi dalam modul-modul dan lebih memberikan kenyamanan pada programmer.

Untuk mengillustrasikan proses-proses yang saling bekerjasama ini digunakan producer-consumer problem. Producer adalah suatu proses yang menghasilkan informasi yang akan dikonsumsi oleh consumer. Sebagai contoh: program untuk mencetak menghasilkan karakter-karakter yang akan dikonsumsi oleh *printer driver*; *compiler* menghasilkan kode assembly yang akan dikonsumsi oleh *assembler*; *assembler* menghasilkan objek modul yang akan dikonsumsi oleh *leader*. Kerja *producer* dan *consumer* ini harus disinkronisasikan sehingga *consumer* tidak akan meminta item yang belum diproduksi oleh producer. *Unbounded-buffer producer-consumer* problem tidak menggunakan batasan ukuran di buffer. *Consumer* dapat selalu meminta item baru, dan *producer* dapat selalu menghasilkan item-item baru. Permasalahan terjadi pada *bounded-buffer producer-consumer* dimana buffer yang digunakan mempunyai ukuran tertentu. *Consumer* harus menunggu jika buffer kosong, dan *producer* harus menunggu jika buffer penuh. Penyelesaian permasalahan *bounded-buffer producer-consumer* dengan solusi shared memory menggunakan data shared berikut :

```
#define BUFFER_SIZE 10
Typedef struct {
   . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Proses producer :

```
item nextProduced;
while (1) {
   while (((in + 1) % BUFFER_SIZE) == out)
         ; /* do nothing */
   buffer[in] = nextProduced;
   in = (in + 1) % BUFFER_SIZE;
}
```

Proses consumer :

```
item nextConsumed;
while (1) {
   while (in == out)
         ; /* do nothing */
   nextConsumed = buffer[out];
   out = (out + 1) % BUFFER_SIZE;
}
```

Variabel `in` dan `out` diinisialisasikan dengan nilai nol. Buffer yang digunakan secara bersama-sama diimplementasikan sebagai larik sirkular dengan 2 pointer logika: `in` dan `out`. Variabel `in` menunjukkan posisi kosong berikutnya pada buffer; `out` menunjukkan posisi penuh pertama pada buffer. Buffer kosong jika `in == out` dan buffer penuh jika `in = (in + 1) % BUFFER_SIZE = out`.

3.5 KOMUNIKASI ANTAR PROSES (INTERPROCESS COMMUNICATION)

Komunikasi antar proses adalah mekanisme proses-proses untuk berkomunikasi dan melakukan sinkronisasi aksinya. Komunikasi dilakukan dengan sistem pesan, dimana proses berkomunikasi dengan proses lain tanpa menggunakan variabel yang di-share. Fasilitas *interprocess communication* (IPC) terdiri dari dua operasi :

send(*pesan*) dimana ukuran pesan bisa tetap atau berbeda-beda

receive(*pesan*)

Apabila proses *P* dan proses *Q* akan melakukan komunikasi, maka kedua proses ini memerlukan :

- Tersedia saluran komunikasi antara kedua proses tersebut.
- Menukar pesan menggunakan send atau receive

Sedangkan implementasi saluran komunikasi dalam bentuk :

- Fisik , misalnya shared memory, hardware bus
- Logika, misalnya properti logika

Implementasi saluran komunikasi harus dapat menjawab pertanyaan berikut :

- Berapa sambungan yang tersedia ?
- Dapatkan sambungan dihubungkan dengan lebih dari dua proses ?
- Berapa banyak sambungan yang terdapat diantara setiap pasangan proses yang berkomunikasi ?
- Bagaimana kapasitas sambungan ?
- Apakah ukuran pesan pada sambungan tetap atau berbeda ?
- Apakah sambungan berupa unidirectional atau bidirectional ?

Terdapat dua bentuk komunikasi antar proses yaitu komunikasi langsung (*direct communication*) dan komunikasi tak langsung (*indirect communication*).

3.5.1 Komunikasi Langsung

Bentuk komunikasi langsung adalah proses melakukan komunikasi langsung ke proses lain. Pada komunikasi langsung, harus disebutkan nama proses secara eksplisit.

send*(P,pesan);* mengirim pesan ke proses *P*.

receive*(Q,pesan);* menerima pesan dari proses *Q*.

Properti yang harus terdapat pada saluran komunikasi terdiri dari :

a. Terdapat sambungan yang dapat bekerja secara otomatis antara tiap pasangan proses yang ingin berkomunikasi.
b. Sambungan tersebut menghubungkan tepat satu pasangan proses yang akan berkomunikasi.
c. Antar tiap-tiap pasangan proses terdapat tepat satu saluran.
d. Sambungan tersebut mungkin bersifat *unidirectional*, namun biasanya *bidirectional*.

3.5.2 Komunikasi Tak Langsung

Pada komunikasi tak langsung pengiriman atau penerimaan pesan dilakukan melalui *mailbox* (port). Mailbox adalah suatu objek yang mana pesan-pesan ditempatkan oleh proses atau dapat dihapus. Tiap-tiap *mailbox* memiliki identitas unik. Dua buah proses dapat saling berkomunikasi hanya jika mereka saling menggunakan mailbox secara bersama-sama.

Properti yang harus disediakan pada saluran komunikasi adalah :

a. Sambungan antara 2 proses diberikan jika antara kedua proses tersebut saling menggunakan mailbox secara bersama-sama.
b. Sambungan tersebut dihubungkan dengan beberapa proses.
c. Antar tiap-tiap pasangan proses yang saling berkomunikasi, ada sejumlah sambungan yang berbeda, tiap-tiap link berhubungan dengan satu mailbox.
d. Sambungan tersebut mungkin bersifat *unidirectional*, namun biasanya *bidirectional*.

Operasi yang terdapat pada system mailbox adalah membuat mailbox baru, mengirim dan menerima pesan melalui mailbox dan menghapus mailbox. Primitif yang terdapat pada komunikasi tak langsung adalah :

send*(A,pesan);* mengirim pesan ke mailbox *A*.

receive*(A,pesan);* menerima pesan dari mailbox *A*.

Perhatikan contoh berikut ini. P_1, P_2 dan P_3 menggunakan mailbox *A* bersama-sama. P_1 mengirim pesan sedangkan P_2 dan P_3 menjalankan operasi **receive**. Proses mana yang mendapatkan pesan. Solusi untuk permasalahan ini dapat menggunaikan salah satu cara di bawah ini :

- Mengijinkan satu sambungan dihubungkan dengan paling banyak dua proses.
- Mengijinkan hanya satu proses pada satu waktu mengeksekusi operasi **receive.**
- Mengijinkan sistem untuk memilih penerima tertentu. Proses pengirim diberitahukan proses mana yang menerima.

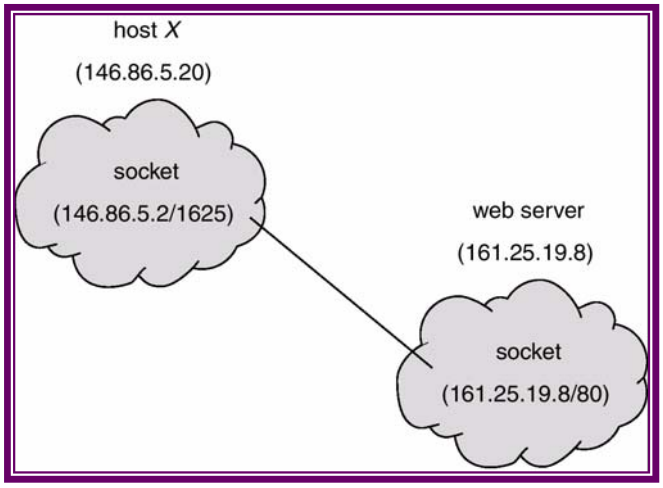
Sambungan mempunyai beberapa kapasitas yang menentukan jumlah pesan yang dapat ditampung sementara. Bentuknya berupa antrian pesan yang dilewatkan ke sambungan. Terdapat tiga cara implementasi antrian pesan tersebut yaitu :

a. *Zero Capacity*: Antrian memiliki panjang maksimum nol, sehingga tidak ada pesan yang menunggu di link. Pada kasus ini, pengirim pesan harus menunggu penerima pesan menerima pesan yang disampaikan sebelum ia mengirim pesan lagi. Kedua proses ini harus berjalan secara sinkron. Sinkronisasi ini sering disebut dengan istilah *rendezvous*.
2. *Bounded Capacity*. Antrian memiliki panjang tertentu (*n*), sehingga ada paling banyak *n* pesan yang menunggu di link. Jika antrian tidak dalam keadaan penuh, maka jika ada pesan baru dapat menempati antrian yang paling akhir, sehingga pengirim tidak perlu menunggu lagi untuk melanjutkan eksekusi. Jika antrian dalam keadaan penuh, maka pengirim harus menunggu sampai ada tempat kosong.
3. *Unbounded Capacity*. Antrian memiliki panjang yang tidak tertentu, sehingga ada sejumlah pesan yang dapat menunggu di link. Pengiriman tidak pernah menunda pekerjaan.

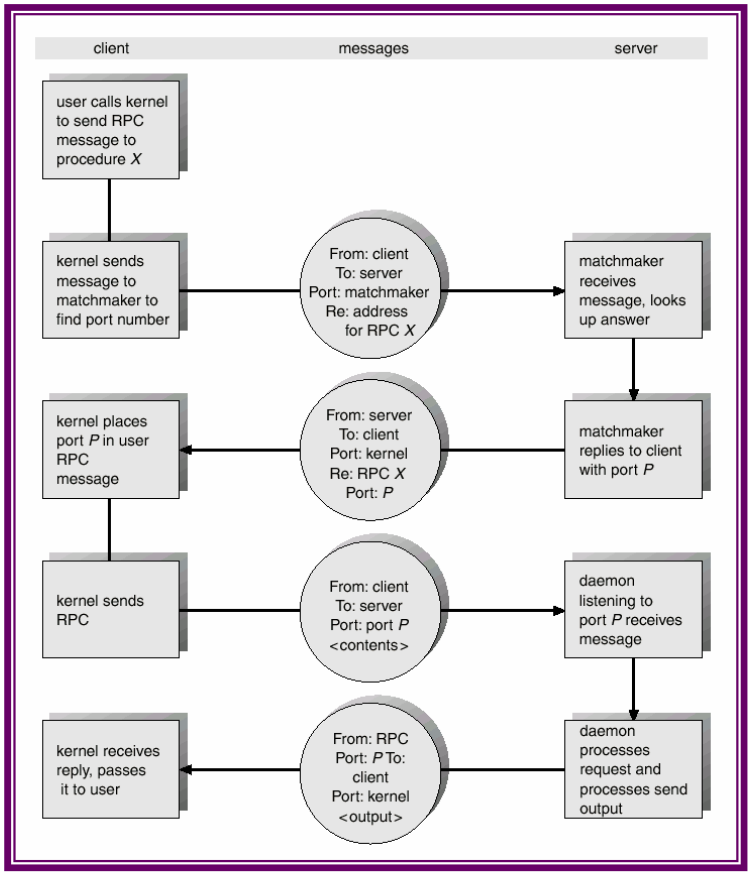
Contoh komunikasi antar proses adalah pada system client server. Komunikasi client server menggunakan berbagai bentuk antara lain socket, *remote procedure call* (RPC) dan *remote method invocation* (RMI).

Sebuah socket didefinisikan sebagai *endpoint for communication*. Socket didefinisikan dengan gabungan antara alamat IP dan port, misalnya socket

161.25.19.8:1625 mengacu ke port **1625** pada host **161.25.19.8**. Komunikasi yang dilakukan terdiri dari sebuah pasangan socket (Gambar 3-7).



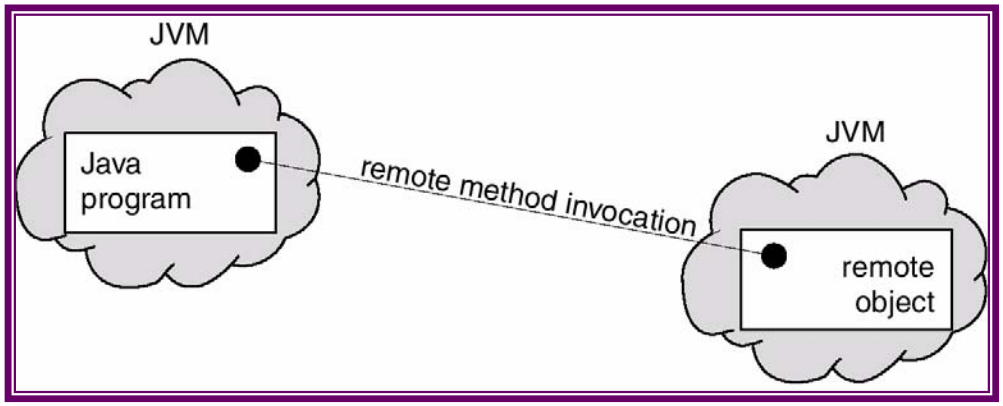
Gambar 3-7: *Komunikasi dengan socket*



Gambar 3-8: *Komunikasi dengan RPC*

Remote procedure call (RPC) merupakan prosedur pemanggilan abstrak antar proses-proses pada system jaringan. Pada RPC terdapat **stub** yaitu proxy pada sisi client untuk prosedur aktual ke server. Stub pada sisi client menghubungi server dan melewatkan parameter, kemudian stub pada sisi server menerima pesan tersebut, menerima parameter dan membentuk prosedur untuk proses server (Gambar 3-8).

Pada bahasa pemrograman Java terdapat *remote method invocation* (RMI) yang merupakan mekanisme untuk berkomunikasi pada jaringan yang mempunyai bentuk yang sejenis dengan RPC. RMI memungkinkan program Java pada satu mesin mengirim dan menerima method dari obyek secara remote (Gambar 3-9).



Gambar 3-9: *Komunikasi dengan RMI*

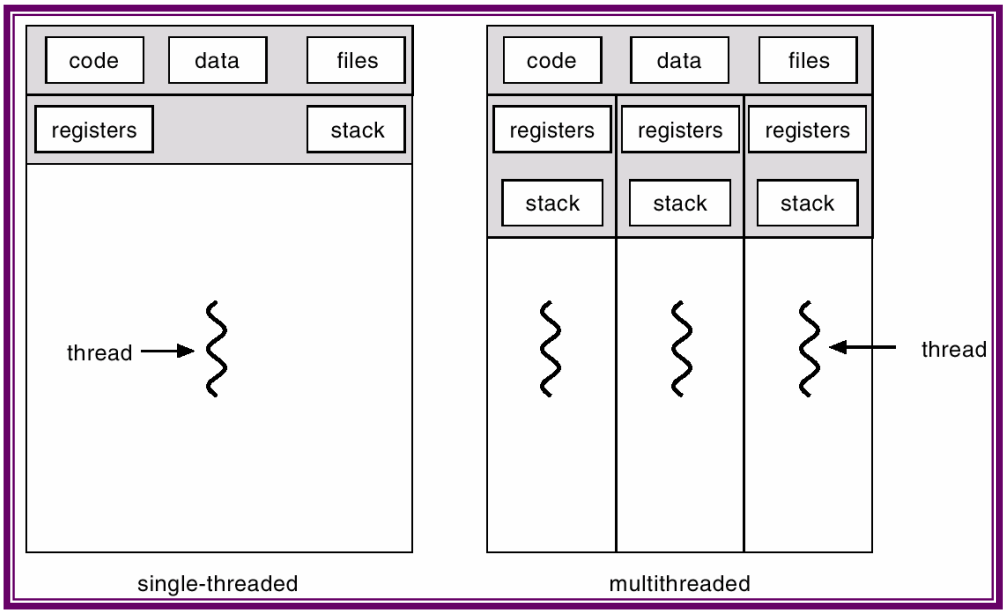
3.6 THREAD

Suatu proses didefinisikan oleh *resource* yang digunakan dan lokasi yang akan dieksekusi. Namun adakalanya proses-proses akan menggunakan resource secara bersama-sama. Suatu unit dasar dari CPU utilization yang berisi program counter, kumpulan register, dan ruang stack disebut dengan *thread* atau *lighweight process* (*LWP*). *Thread* akan bekerjasama dengan *thread* yang lainnya dalam hal penggunaan bagian kode, bagian data, dan *resource* sistem operasi, seperti open file dan sinyal secara kolektif yang sering disebut dengan *task.*

Apabila dilakukan perbandingaan antara sistem *multi thread* dengan sistem multi proses dapat disimak berikut. Pada multi proses, setiap proses secara terpisah

melakukan operasi tidak bekerja sama dengan proses lain, setiap proses mempunyai *program counter*, *stack register* dan ruang alamat sendiri. Organisasi jenis ini berguna jika *job* dibentuk oleh proses-proses yang tidak saling berhubungan. Multi proses membentuk *task* yang sama. Sebagai contoh, multi proses dapat menyediakan data untuk mesin secara *remote* pada implementasi system file jaringan. Hal ini lebih efisien apabila satu proses terdiri dari *multi thread* melayani tugas yang sama. Pada implementasi multi proses, setiap proses mengeksekusi kode yang sama tetapi mempunyai memori dan *resource file* sendiri. Satu proses *multi thread* menggunakan *resource* lebih sedikit daripada multi proses, termasuk memory, *open file* dan penjadwalan CPU.

Seperti halnya proses, *thread* memiliki status: *ready*, *blocked*, *running* dan *terminated*, dan hanya satu *thread* yang aktif dalam satu waktu. *Thread* dapat membuat child *thread*. Jika satu *thread* dalam keadaan *blocked*, maka *thread* yang lainnya dapat dijalankan. Namun, tidak saling bebas, Sebab semua *thread* dapat mengakses setiap alamat dalam satu task, *thread* dapat membaca dan menulisi stack dari *thread* yang lainnya. Sehingga tidak ada proteksi antara satu *thread* terhadap *thread* yang lainnya. Suatu proses dapat terdiri dari satu thread (*single thread*) dan beberapa thread (*multi thread*) seperti pada Gambar 3-10.

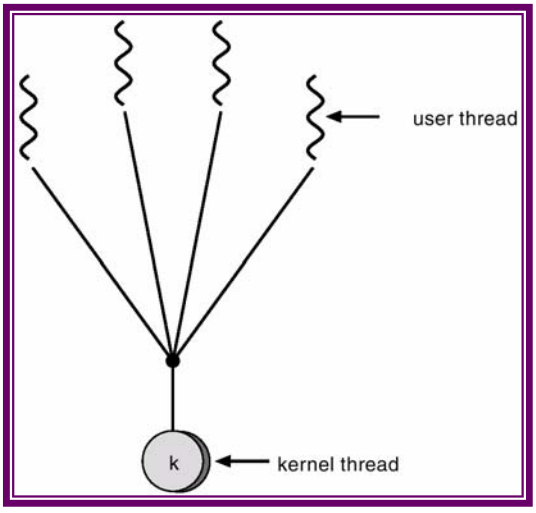


Gambar 3-10: *single thread dan multi thread*

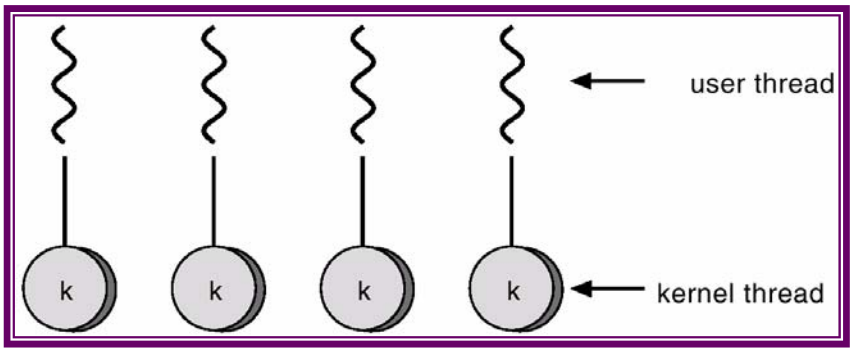
Keuntungan sistem *thread* adalah pada respon lebih cepat, menggunakan *resource* bersama-sama, lebih ekonomis dan meningkatkan utilitas arsitektur mikroprosessor.

Thread terdiri dari dua bentuk yaitu *user thread* dan *kernel thread*. *User thread* adalah *thread* yang diatur dengan menggunakan pustaka *user level thread*. Contoh sistem yang menggunakan *user thread* adalah POSIX *Pthreads*, Mach *C-threads* dan Solaris *threads*. Sedangkan *kernel thread* adalah *thread* yang didukung oleh Kernel. Contoh sistem yang menggunakan *kernel thread* adalah Windows 95/98/NT/2000, Solaris, Tru64 UNIX, BeOS dan Linux.

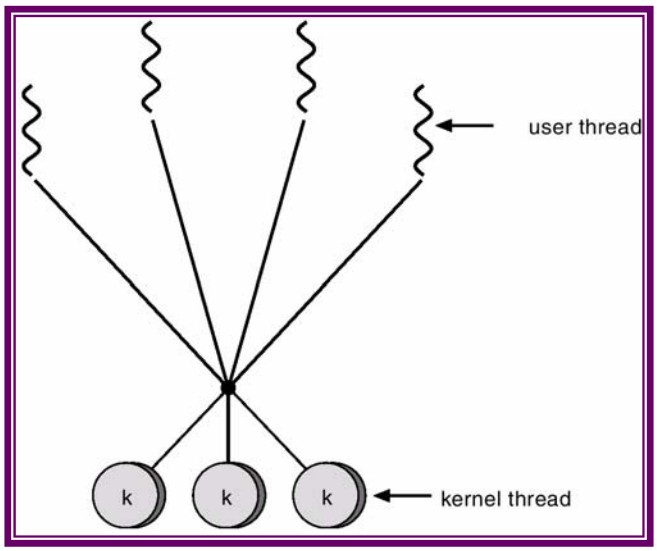
Model *multi thread* terdiri dari model *Many-to-One*, *One-to-One* dan *Many-to-Many*. Pada model *Many-to-One* , beberapa *user level thread* dipetakan ke satu *kernel thread* dan digunakan pada sistem yang tidak mendukung *kernel threads* seperti pada Gambar 3-11. Pada model *One-to-One*, setiap *user-level thread* dipetakan ke *kernel thread* seperti pada Gambar 3-12, misalnya pada Windows 95/98/NT/2000 dan OS/2. Pada model *Many-to-Many*, *user level thread* dipetakan ke beberapa *kernel threads*. Pada sistem operasi ini akan dibuat sejumlah *kernel thread* seperti Gambar 3-13, contohnya Solaris 2 dan Windows NT/2000 dengan *ThreadFiber* package.



Gambar 3-11: *model Many to One*

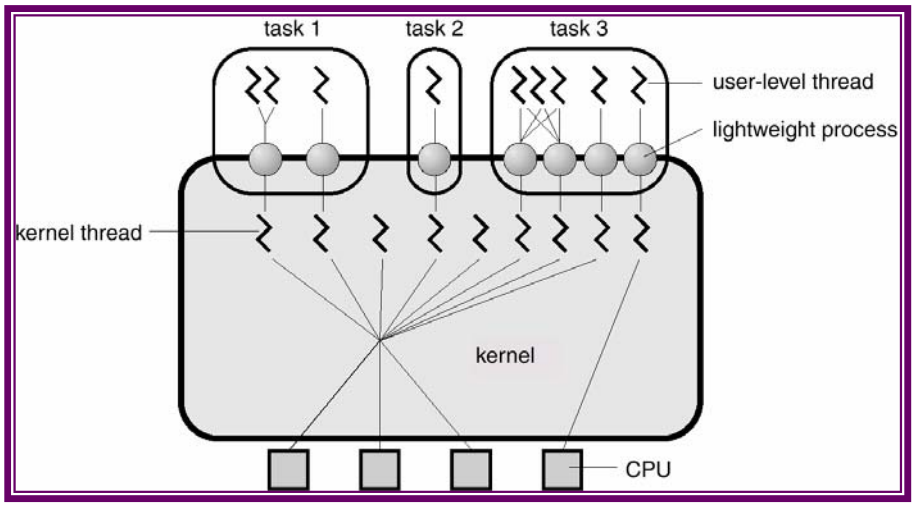


Gambar 3-12: *model One to One*



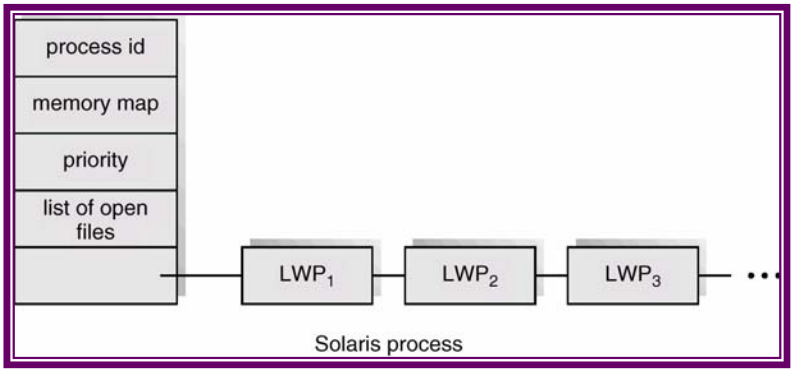
Gambar 3-13: *model Many to Many*

Contoh sistem operasi yang menggunakan sistem *thread* adalah Solaris 2. Solaris 2 mendukung *user-level thread* dan *kernel thread.* Pembuatan dan penjadwalan *user level thread* didukung oleh pustaka dan kernel tidak mempunyai pengetahuan tentang *user level thread*. Antara *user level thread* dan *kernel thread* terdapat perantara yang disebut dengan *lightweight process* (LWP). Setiap *task* terdapat setidaknya satu LWP seperti pada Gambar 3-14. Semua operasi pada kernel dieksekusi oleh *kernel level thread*. Terdapat satu *kernel level thread* untuk setiap LWP dan terdapat beberapa *kernel level thread* yang menjalankan kernel baik yang dihubungkan maupun tidak dengan LWP.



Gambar 3-14: *Thread pada Solaris 2*

Kernel thread mempunyai struktur data yang kecil dan sebuah *stack*. Perpindahan antar *kernel thread* tidak mengubah informasi akses memori sehingga relatif cepat. LWP berisi *process control block* dengan register data, informasi akutansi dan informasi memori. Perpindahan antar LWP membutuhkan tambahan pekerjaan dan relatif lambat. *User level thread* hanya memerlukan sebuah *stack* dan sebuah *program counter*, tanpa *resource* dari kernel. Kernel tidak dilibatkan dalam penjadwalan *user level thread*, sehingga perpindahan antar *user level thread* sangat cepat. Terdapat ratusan *user level thread*, tetapi semua kernel yang terlihat dalam bentuk LWP yang mendukung *user-level thread*. Proses *thread* pada Solaris 2 dapat dilihat pada Gambar 3-15.

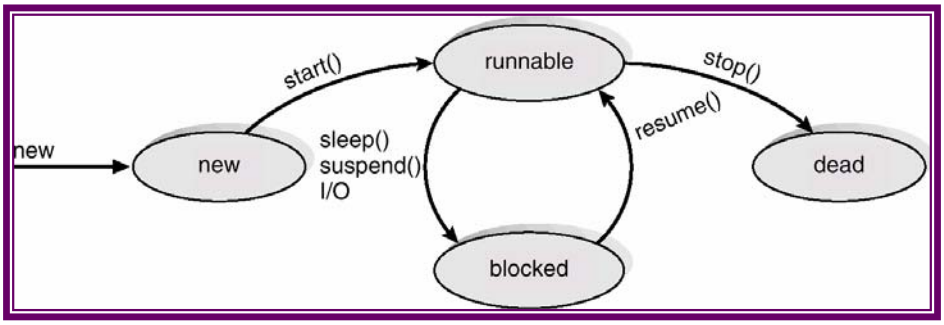


Gambar 3-15: *Proses Solaris 2*

Thread pada Windows 2000 mengimplementasikan pemetaan *one-to-one*. Setiap thread terdiri dari *thread id*, kumpulan *register, stack user* dan kernel yang terpisat serta ruang data privat.

Thread pada Linux sering disebut sebagai *task* daripada *thread*. Pembuatan *thread* dilakukan dengan menggunakan *system call* **clone()**. **Clone()** memungkinkan *task* anak menggunakan ruang alamat dari *task* (proses) *parent*.

Bahasa pemrograman Java menggunakan *Java thread* yang dibuat dengan menggunakan *class Thread* dan mengimplementasikan antar muka yang bersifat r*unnable* (dapat dijalankan). *Java thread* diatur oleh *Java virtual machine* (JVM). *Java thread* terdiri dari *state new, runnable, blocked* dan *dead* seperti Gambar 3-16.



Gambar 3-16: *State pada Java Thread*

LATIHAN SOAL :

1. Sebutkan *state* pada proses dan jelaskan diagram proses
2. Apa yang dimaksud *short term scheduler* dan *long term scheduler* ?
3. Jelaskan 4 alasan mengapa proses harus bekerja sama.
4. Tuliskan kode program untuk penyelesaian permasalahan producer consumer dengan menggunakan *shared memory.*
5. Diketahui skema komunikasi antar proses menggunakan mailbox
    a. Proses *P* ingin menunggu 2 pesan, satu dari mailbox *A* dan satu dari mailbox *B.* Tunjukkan urutan **send** dan **receive** yang dieksekusi

b. Bagaimana urutan **send** dan **receive** yang dieksekusi *P* jika *P* ingin menunggu satu pesan dari mailbox *A* atau mailbox *B* (salah satu atau keduanya)

6. Jelaskah apa yang dimaksud dengan *thread* dan struktur dari *thread*.
7. Jelaskan empat keuntungan menggunakan *threads* pada *multiple process*.
8. Apakah perbedaan antara *user-level thread* dan *kernel-supported threads* ?
9. Ada 3 model *multithreading*, jelaskan.
10. Jelaskan state pada *Java thread*.