JURNAL TEKNOLOGI TECHNOSCIENTIA ISSN: 1979-8415
Vol. 5 No. 2 Februari 2013

CRITICAL REGION - SEMAPHORE MUTEX DALAM CPU OS SIMULATOR V.7.2.27.

Sp. Hariningsih[1]

[1]Program Studi Sistem Informasi, Universitas Dian Nuswantoro Semarang

Masuk: 6 Nopember 2012, revisi masuk : 8 Desember 2012, diterima: 21 Januari 2013

ABSTRACT

Operating system is a in computer program that regulates the resourcein a computer . The operating systems should monitor the status of each resource, decide which process has to get those resources, allocate resources, and claim back when it has done. To implement the process model, the operating system use a table / array called table with one entry for each process. Each entry contains the status of the process, the program counter, stack pointer, memory allocation, file status, scheduling information, etc. Wich from employment to ready status. In multiprogramming system, the processor also switch from one program to another program, run the program in a few milliseconds. At any given time, the processor is actually just did a program, but in a second of accumulation time, the processor was working on several programs, so the program look run in parallel. Critical region is a port of a program being access memory accordingly. Critical region happere on a mutual exclussion process, which is only one process being execute. Neverthless, there is a certain that we call race condition. It is a situation where some processes are access and manipulate the data accordingly. Thos, we have to pind some methode to prevent leftover process in writing process and reading process for data shared accordingly, at is mutual exclussion. It will examined on OS CPU simulator. Althought it can prevent the race conditions, it isn't enough to do cooperation between some proces in parallel efficiently in using data shared.

Keywords : Critical Region/Section, Process, Mutual Exclussion, Semaphore, Storage.

INTISARI

Sistem operasi adalah program komputer yang mengatur *resource-resource* pada komputer. Sistem Operasi harus mengawasi status dari setiap sumber daya, memutuskan proses mana yang harus mendapatkan sumber daya tersebut, mengalokasikan sumber daya, dan meng-klaim kembali jika sudah selesai. Untuk mengimplementasikan model proses, sistem operasi menggunakan suatu tabel / *array* yang disebut tabel proses dengan 1 *entry* per-*proses*. Setiap *entry* berisi tentang status proses, program *counter*, *stack pointer*, alokasi memori, status file, informasi *schedulling* / penjadwalan informasi, dll dari status kerja ke status siap. Dalam sistem *multiprogramming*, prosesor juga beralih dari satu program ke program yang lain, menjalankan program tersebut beberapa milidetik. Pada suatu waktu tertentu, prosesor sebenarnya hanya melakukan satu program, tetapi dalam akumulasi waktu satu detik, prosesor terasa bekerja pada beberapa program, sehingga memberikan kesan pada pemakai bahwa beberapa program dilakukan secara paralel. *Critical region/critical section* merupakan bagian dari program yang sedang mengakses memori secara bersama. *Critical region* terjadi pada proses *mutual exclussion*, yaitu jaminan hanya satu proses yang sedang dilakukan eksekusi. Jika terjadi *race condition* yaitu situasi proses mengakses dan memanipulasi data secara bersamaan harus ditemukan beberapa jalan untuk mencegah lebih dari satu proses untuk melakukan proses *writing* dan *reading* kepada *shared* data pada saat yang sama yaitu dengan *mutual ecxlussion*. Hal ini akan di uji cobakan pada OS CPU simulator. Walaupun dapat mencegah *race condition*, tetapi tidak cukup untuk melakukan kerjasama antar proses-proses secara paralel dengan baik dan efisien dalam menggunakan *shared* data.

Kata kunci : *Critical Region/Section, Process, Mutual Exclussion, Semaphore, Storag*

[1]*aningsp@gmail.com*

PENDAHULUAN

Sistem komputer selama ini selalu dikembangkan berkaitan dengan pengembangan ilmu dan teknologi pengetahuan. Hal ini berdampak pada sistem operasi yang selalu mengikuti perkembangan perangkat keras dan perangkat lunak. Komputer merupakan media bantu penyelesai masalah dari pengguna. Permasalahan tersebut diselesaikan oleh prosesor komputer yang kita tunggu-tunggu sebagai modal dalam pengambilan keputusan yaitu data dan informasi. Peran sistem operasi di dalam sistem komputer berfungsi sebagai pengendali di antara sumberdaya sumberdaya salah satunya adalah sumberdaya pemroses. Sejauh ini pengguna mempunyai anggapan bahwa saat komputer melakukan eksekusi terhadap proses-proses, komputer tersebut hanya melakukan eksekusi terhadap satu proses dan satu masalah saja. Hal ini tidak benar, prosesor akan melaksanakan eksekusi kepada antrian proses secara adil.

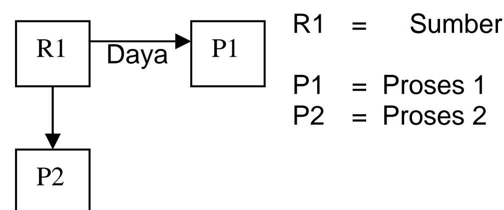
Jika dipandang dari segi pengguna mempunyai anggapan bahwa dirinya sendiri yang sedang dieksekusi dan tidak ada proses lain yang sedang dikerjakan dalam prosesor. Anggapan ini salah, dengan adanya sistem operasi yang berfungsi sebagai pengontrol dan pengendali prosesor maka semua antrian proses dilayani secara adil dan seksama sesuai dengan jatah waktu eksekusinya. Proses-proses akan dieksekusi secara bersama dalam jeda waktu milidetik saja. Anggapan pengguna ini sering disebut dengan *pseudoparalellisme*, karena cepatnya akses tersebut. Sistem operasilah yang berfungsi sebagai kendali diantara proses-proses sehingga sistem operasi harus mengawasi status dari setiap sumber daya, menetapkan alokasi dan dealokasi sumberdaya kepada proses-proses yang membutuhkan.

METODE

Konsep penting dalam sistem operasi adalah status "proses" yaitu abstraksi dari jalannya program. Saat ini komputer diciptakan dapat melakukan beberapa proses pada saat yang bersamaan, misalnya pada saat tertentu komputer mengeksekusi program dari pengguna, komputer dapat juga membaca dari disk, mencetak ke printer atau ke layar bahkan saat itu juga sedang mengambil informasi dari internet sekalipun, tetapi bagaimana dengan terjadinya redudansi data, inkonsistensi data ataupun terjadi *race condition.*

Kunci untuk mencegah masalah proses bersama yang melibatkan *shared* memori, *shared* berkas, *and shared* sumber daya yang lain adalah menemukan beberapa jalan untuk mencegah sehingga tidak terjadi proses *writing* dan *reading* kepada *shared* data pada saat yang sama yaitu mutex (*mutual exclussion*). *Critical section* merupakan bagian program yang sedang mengakses memori atau sumber daya yang di pakai bersama. Hal ini akan dijelaskan dengan penggunaan 2 *thread* pada saat yang bersamaan pada sistem dengan *semaphore mutex* CPU OS simulator V.7.2.27.

Proses dikatakan konkuren apabila proses-proses tersebut secara bersama-sama melakukan eksekusi sumberdaya komputer, baik sumberdaya memori, sumberdaya informasi, maupun sumberdaya device. Jadi bisa disimpulkan proses mempunyai sifat konkuren adalah suatu proses dimana lebih dari satu proses dapat dieksekusi oleh sumber daya komputer dalam waktu yang bersamaan. Digambarkan seperti pada Gambar 1.

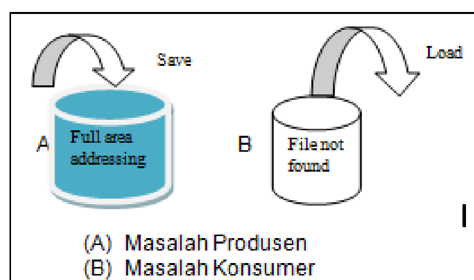


Gambar 1. Proses Konkurensi

Tetapi proses konkuren ini mempunyai beberapa permasalahan, antara lain adalah proses *mutual exclussion*. Proses *mutual exclussion* adalah jaminan bahwa hanya dirinya sendirilah yang dieksekusi oleh sistem komputer. Bisa disimpulkan bahwa

proses *mutual exclussion* adalah *single user*, karena hanya satu proses saja yang dapat menggunakan sumber daya dan proses lain menunggu untuk dieksekusi setelah proses lain telah selesai dieksekusi.

Proses Sinkronisasi yaitu proses tunggal yang mempunyai masalah proses produsen dan masalah proses konsumer. Masalah produsen terjadi apabila proses-proses akan menciptakan proses, tetapi di dalam memori tersebut mempunyai informasi kapasitas memori penuh sehingga memori tersebut tidak akan bisa menampung berkas lagi. Hal ini dapat diselesaikan dengan penciptaan *buffer* (penyangga), tetapi akan menimbulkan masalah baru yaitu *multiprogramming* dengan sistem *swapping*. Masalah konsumer terjadi apabila pada saat proses pengalokasian data kepada user tetapi file data tersebut tidak tersedia dalam artian memori *"file not found"*. Dua macam masalah ini hanya dapat diselesaikan dengan metode sinkronisasi.



Gambar 3. Masalah proses sinkronisasi

Metode Sinkronisasi merupakan suatu metode untuk menyelesaikan masalah pada proses *sinkronisasi*, yaitu masalah produsen dan masalah konsumen. Metode ini untuk menjaga ketersediaan data atau berkas didalam memori apabila terdapat user yang ingin menggunakan sumber daya informasi dan akan menjaga ketersediaan memori yang kosong apabila user ingin menciptakan proses dalam memori tersebut.

Dengan kata lain metode sinkronisasi bertugas menjaga memori dalam keadaan siap untuk dialokasikan dan didealokasikan kepada para pemakainya. Oleh karena itu diperlukan metode sinkronisasi untuk menjaga kondisi memori tersebut.

Semaphore termasuk pendekatan yang diajukan oleh Djikstr auntuk digunakan untuk memonitor dan mengontrol ketersediaan sumberdaya sistem, seperti halnya pembagian segmen memori (Tanembaum, 1992). Seperti proses yang sedang aktif dapat berhenti pada suatu saat (seperti halnya pada penjadwalan proses *Round Robin* yang dapat dihentikan oleh nilai *quantum*nya), sampai proses mendapatkan sinyal interupsi untuk melanjutkan eksekusinya. Semaphore mempunyai dua sifat (Wiliiam Stelling, 2011), yaitu: Semaphore dapat mempunyai variabel dengan nilai non-negatif dan terdapat dua operasi terhadap semaphore, yaitu Down dan Up. Usulan asli yang disampaikan Djikstra adalah operasi P dan V.

Operasi Down (P). Operasi ini menurunkan nilai semaphore, jika nilai semaphore menjadi non-positif maka proses yang mengeksekusinya *diblocked.* Operasi *down* adalah atomic, dalam artian proses dalam status aktif tak dapat diberhentikan oleh proses lain sampai proses tersebut selesai di eksekusi.

Operasi Up (V). Operasi Up menaikkan nilai semaphore yang terdapat dalam antrian proses. Urutan proses yang akan dieksekusi dipilih secara acak.

Semaphore adalah sinyal interupsi yang digunakan untuk memeriksa apakah sumber daya saat ini sedang digunakan oleh proses (*thread*). Misalnya, jika suatu proses ingin menggunakan sumber daya outputan, terlebih dahulu perlu memastikan apakah peralatan pencetak (*outputan*) sedang dalam status *off* atau *on*. Jika *device* tersebut dalam keadaan *on* maka sinyal interupsi akan memberikan kode penundaan untuk melakukan akses pada proses yang antri untuk di lakukan pencetakan, sampai peralatan pencetakan mengirimkan sinyal interupsi off pada antrian proses baru dapat

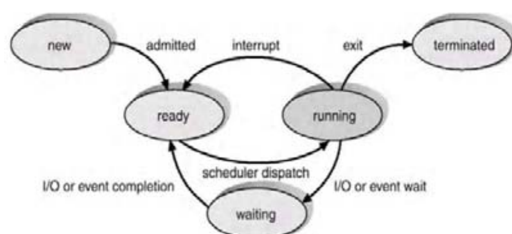
dilakukan eksekusi. Tetapi jika peralatan outputan dalam status *off*, maka *output device* akan mengirimkan sinyal interupsi kepada antrian proses yang berarti peralatan outputan siap melaksanakan perintah pencetakan (Sp. Hariningsih, 2003).

Proses *Multiprogramming* pada tingkat perangkat keras, melakukan proses satu persatu secara bergantian dalam waktu yang sangat cepat atau bersamaan, karena setiap proses mempunyai satu CPU maya.

Proses Pseudoparallelism. Pada level tingkat pengguna, melakukan lebih dari satu pekerjaan dalam waktu yang bersamaan. Secara konsep setiap proses mempunyai satu CPU maya, tetapi pada kenyataannya dalah *multiprogramming*. Maka akan lebih mudah menganggap kumpulan proses yang berjalan secara parallel.

PEMBAHASAN

Hubungan state dasar proses dalam pemroses dapat digambarkan seperti tampak pada Gambar 4. (Bambang Hariyanto, 1997)



Gambar 4. Diagram state dasar proses

Keterangan:

New : Proses baru yang masuk dalam antrian proses
Ready : Proses menunggu untuk dieksekusi
Running : Eksekusi diantara antrian proses
Terminated : Proses yang telah selesai dieksekusi dengan status rampung secara sempurna
Waiting : Proses yang menunggu untuk dieksekusi karena proses yang terdapat status processor belum selesai dengan sempurna sehingga proses memerlukan kontrol inputan kembali.
Dispatch : Status proses beralih ke interupsi peralatan keluaran
Interupt/ Time out : Status pemroses yang meminta untuk beralih ke proses lain karena proses telah selesai dilaksanakan. Ataupun proses diberhentikan oleh pemroses secara paksa (seperti pada model round robin) dan beralih ke proses lain.

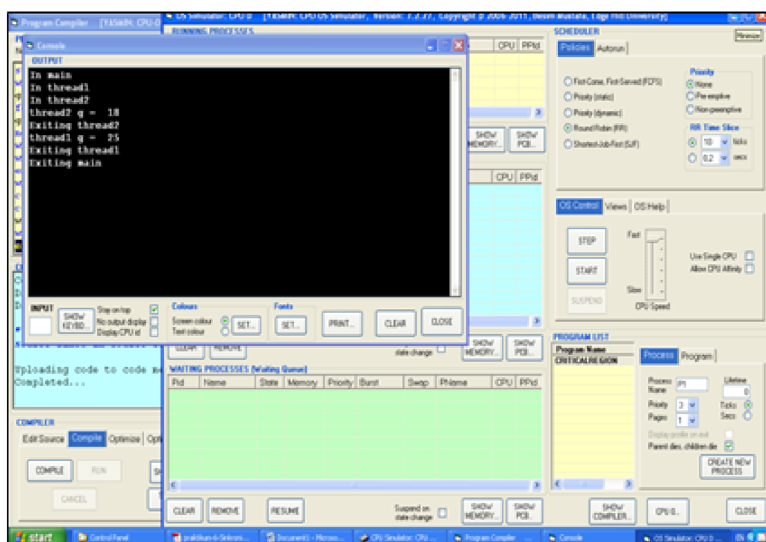
Dengan menggunakan aplikasi CPU-OS Simulator program *critical region*, hal ini akan membuat dua *thread* yaitu *thread1* dan *thread2*. Masing-masing *thread* akan menaikkan nilai variabel global g dalam tiap *loop*. Yang termasuk syncronisasi yaitu mulai dari Ladd no 000,0006,0012,0016,0020. Proses tersebut dapat dilihat pada Gambar 5.

Pada modifikasi yang pertama ditambahkan kata kunci *synchronise*. Pemberian tambahan kunci sinkronisasi pada komunikasi diantara proses-proses membutuhkan *place by calls* untuk mengirim dan menerima data *primitive* . Proses tersebut dapat dilihat pada Gambar 6. Terdapat *design* yang berbeda-beda dalam implementasi setiap *primitive*. Pengiriman pesan dapat dilakukan dengan :[1].Pengiriman yang diblok : Proses pengiriman data akan diblok sampai pesan diterima oleh proses penerima (*receiving process*) atau oleh *mailbox*. [2]. Pengiriman yang tidak diblok : Proses pengiriman pesan dan mengkalkulasi operasi. [3]. Penerimaan yang diblok : Penerima memblok sampai pesan tersedia. [4]. Penerimaan yang tidak diblok : Penerima mengembalikan pesan valid atau null.

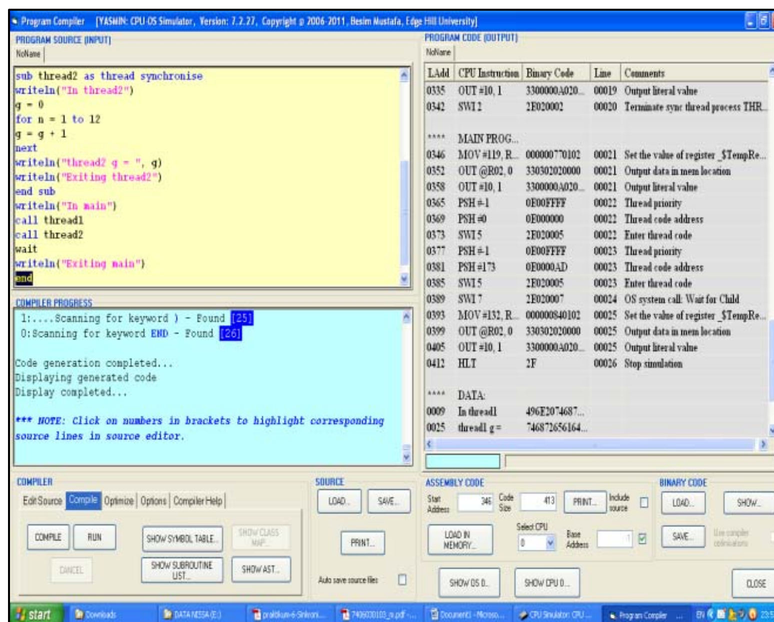
Critical regions seringkali diimplementasikan menggunakan *semaphore* dan *mutex. Semaphore* adalah pendekatan yang dikemukakan Djikstra. Prinsip semaphore adalah Dua proses atau lebih dapat bekerja sama dengan menggunakan penanda-penanda sederhana. Proses dipaksa berhenti sampai proses memperoleh penanda tertentu. Sembarang kebutuhan

 ISSN: 1979-8415

koordinasi kompleks dapat dipenuhi dengan strukstur penanda yang sesuai kebutuhannya. Variabel khusus untuk penandaan ini disebut *semaphore*. *Semaphore* adalah alat untuk sinkronisasi yang tidak membutuhkan *busy waiting*. *Semaphore S(wait)* berupa variable integer. *Semaphore* hanya dapat diakses melalui operasi atomik yang tak dapat diinterupsi sampai kode selesai.

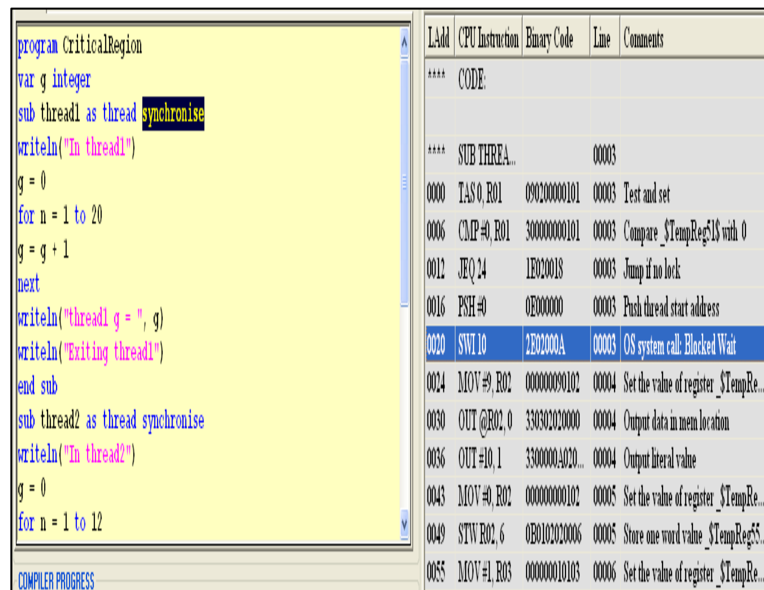


Gambar 5. Running Process Round Robin



Gambar 6. Program source input pada CPU-OS simulator

 ISSN: 1979-8415



Gambar 7. Program code synchronisasi

Pada modifikasi yang kedua digunakan kata kunci *enter* dan *leave.* Pada system komputer terdapat sumberdaya yang tidak dapat dipakai bersama pada saat yang bersamaan seperti pada penggunaan *peralatan pencetak*, Sumberdaya seperti hanya dapat menjalankan satu proses pada suatu saat, sumber daya ini disebut sumber daya kritis. Program yang menggunakan sumber daya kritis disebut sedang memasuki *critical section.*

Sistem operasi memberikan fasilitas untuk pemrogram dapat memberikan indikasi keberadaan critical region. Sistem operasi menyediakan layanan (berupa system call) untuk mencagah suatu proses masuk kedalam critical region akan tetapi di dalam critical region terdapat proses lain yang sedang berjalan. Mutual exclusion merupakan solusi bagi masalah pada critical region / section.

Ketika ***wait*** dijalankan oleh thread, kita memiliki dua kemungkinan yaitu Penghitung *S* adalah positif, dalam hal ini, counter mengalami penurunan sebesar satu. Kemungkinan kedua Penghitung *S* adalah nol, dalam hal ini,thread ditangguhkan dan dimasukkan ke dalam antrian pribadi *S.*

Ketika **Sinyal** dijalankan oleh thread, kita memiliki dua kemungkinan: . Kemungkinan pertama antrian *S* tidak memiliki *thread* menunggu Penghitung *S* ditingkatkan oleh satu dan proses kembali dieksekusi maka modifikasi yang mungkin adalah S:=S-1.

```
wait(S) : while(S) <= 0 do no-op;
S:=S-1;
Signal(S) : S:=S+1;
```

Antrian *S* telah menunggu *threads*, Dalam hal ini, konter ***S*** harus nol (lihat pembahasan wait di atas). Salah satu proses menunggu akan diizinkan untuk meninggalkan antrian dan melanjutkan pelaksanaannya.

Operasi *wait* adalah atomik. Ini berarti sekali kegiatan *wait* mulai (yaitu, pengujian dan penurunan nilai *counter* dan memasukkan benang ke dalam antrian), mereka akan terus sampai akhir tanpa gangguan apapun. Lebih tepatnya, meskipun ada banyak langkah untuk melaksanakan *wait* dan *Signal*, langkah-langkah ini dianggap sebagai instruksi non-interruptible tunggal. Demikian pula, hal yang sama berlaku untuk Sinyal. Apalagi, jika lebih dari satu proses mencoba mengeksekusi signal, hanya satu dari mereka akan berhasil. Kita tidak

boleh membuat asumsi tentang mana proses yang akan berhasil.

Status *wait* karena dapat menyebabkan thread untuk memblokir *(yaitu,* ketika *counter nol*), ia memiliki efek yang sama dari operasi kunci dari sebuah kunci *mutex*. Demikian pula, sebuah sinyal dapat melepaskan thread wait, dan mirip dengan membuka operasi. Bahkan, *semaphores* dapat digunakan sebagai kunci *mutex*. Pertimbangkan semaphore *S* dengan nilai awal 1. Kemudian, *wait* dan *Signal* untuk mengunci dan membuka.

Perlu diingat bahwa nilai awal *counter* dari *S* adalah 1. Misalkan sejumlah *thread* mencoba untuk eksekusi *wait*. Karena hanya ada satu *thread* berhasil dapat mengeksekusi *wait, thread* ini, katakanlah *A,* menyebabkan *counter* berkurang sebesar 1, dan memasuki bagian yang kritis. Karena nilai awal *counter* adalah 1, sekali *thread A* memasuki *critical section*, konter menjadi 0, dan, sebagai hasilnya semua usaha berikutnya dalam melaksanakan *wait* akan diblokir. Oleh karena itu, wait mirip untuk mengunci.

Ketika Sebuah *thread* keluar dari *critical section*, *Signal* dijalankan. Jika ada menunggu *thread*, salah satu dari mereka akan dirilis, dan *thread* ini dirilis memasuki *critical section*. Perhatikan bahwa *counter* masih nol (karena, dalam hal ini, Sinyal tidak meningkatkan dan status wait tidak mengurangi *counter*), yang berarti semua thread berikutnya yang mencoba mengeksekusi wait diblokir. Di sisi lain, jika tidak ada *thread* menunggu, pelaksanaan Sinyal menyebabkan nilai dari *counter* akan meningkat dengan 1, sehingga nilai saat ini 1. Dalam hal ini, *thread* berikutnya yang mengeksekusi status bisa masuk ke bagian kritis. Oleh karena itu, Sinyal untuk membuka. Singkatnya, pengaturan *counter* untuk 1 awalnya akan menjamin bahwa paling banyak satu *thread* bisa di bagian kritis.

Nilai *counter* adalah 1 atau 0, dan tidak pernah memiliki nilai lain sehingga disebut sebagai *semaphore biner.* Jika diganti *counter* dengan variabel Boolean dan menafsirkan 1 dan 0 sebagai true (kunci terbuka) dan *false* (kunci tertutup), masing-masing, maka semaphore biner menjadi kunci *mutex*. Karena itu kunci *mutex* atau *semaphore* biner bergantian. Proses tersebut dapat dilihat pada Gambar 7.

Berikut contoh nyata untuk suatu *critical region* (atau mutex region).

```
Program Give_File_to_spooler;
Var
in : Integer;
berkasA, berkasB: File;
ProcedureStore (Berkas: File, next_slot:
Integer);
{Untukmenyimpanberkaspadaslot
kenext_slot}
Procedure ProsesA;
Var
next_free_slot: Integer;
Begin
next_free_slot:=in;
store(BerkasA, next_free_slot);
in:=next_free_slot+1;
End;
ProcedureProsesB;
Var
next_free_slot:Integer;
Begin
next_free_slot:=in;
store(BerkasB, next_free_slot);
in:=next_free_slot+1;
End;
```

Beberapa arsitektur komputer memiliki instruksi *"test-and-set"* untuk menerapkan critical region. karena Metode *Test and Set* melakukan *testing* dan memodifikasi isi memori secara atomic, struktur fungsi *Test and Set* sebagai berikut :

```
boolean TestAndSet (boolean &target)
{
boolean rv = target;
tqrget = true;
return rv;
}
```

Untuk menyelesaikan permasalahan *mutual exclusion* dengan metode *Test and Set* maka digunakan variable umum berikut :

boolean lock = false;

Sedangkan Process *Pi* mempunyai struktur sebagai berikut :

do {

```
while (TestAndSet(lock)) ;
critical section
lock = false;
        remainder section
        }
```

Dengan menggunakan OS CPU Simulator V. 7.2.27 maka dapat dibuktikan bahwa kejadian *race condition* pada critical region dapat dihindari (Gambar 7)

KESIMPULAN

Thread mirip seperti *little-mini process*. Setiap *thread* berjalan sekuensial yang mempunyai program *counter* dan *stack* sendiri. *Thread* juga men-*share* CPU seperti proses. *Thread* dalam satu proses menempati *address space* yang sama, tidak ada proteksi penggunaan memori antar *thread* karena proses dimiliki oleh satu user.

Thread dapat berada pada empat *state* yang berbeda, seperti process (*running, blocked, ready, terminated*) Terdapat tiga model process pada server : [1}]*thread* diciptakan untuk dapat melakukan paralelisme yang dikombinasikan dengan eksekusi sekuensial dan *blocking system calls.. [2]. single treads server* menggunakan *blocking system calls*, tetapi kinerja sistem tidak baik [3]. *finite-state machine,* kinerja baik dengan melakukan *parallelisme*, tetapi menggunakan *nonblocking calls*, sehingga sulit dalam memprogram.

Dari definisi semaphore dapat disimpulkan Secara umum, tidak ada cara untuk mengetahui sebelum proses *decrement* sebuah semaphore akan terblokir atau tidak. Setelah proses *increment semaphore* dan proses lain akan berjalan, kedua proses terus berjalan bersamaan. Tidak ada cara untuk mengetahui proses, jika salah satu, akan segera melanjutkan pada sistem prosesor tunggal. Saat signal *semaphore*, Anda tidak perlu tahu apakah proses yang lain sedang menunggu, sehingga jumlah proses yang diblokir mungkin nol atau satu. Prinsip semaphore adalah dua proses atau lebih dapat bekerjasama dengan menggunakan penanda-penanda sederhana. Proses dipaksa berhenti sampai proses tersebut memperoleh penanda tertentu.

Dengan menggunakan OS CPU Simulator V. 7.2.27 maka dapat dibuktikan bahwa kejadian *race condition* pada critical region dapat dihindari .

DAFTAR PUSTAKA

Andrew S. Tanembaun, *Modern Operating System,* Prentice Hall. Englewood Cliffs, New Jersey. 1992.

Bambang Hariyanto, Ir, *Sistem Operasi*. Penerbit Informatika Bandung, cetakan pertama Desember 1997.

Sp. Hariningsih, *Sistem Operasi*. Penerbit Graha Ilmu Yogyakarta, Edisi pertama : 2003, viii+176 halaman. ISBN : 979-3289-24-4

William Stallings, Operating Systems, Fourth Edition, Prentice Hall. 2001

http://docs.linux.cz/

http://linux-tutorial.info/