31

Semafor

Seperti yang kita ketahui sekarang, seseorang membutuhkan variabel kunci dan kondisi untuk menyelesaikan berbagai masalah konkurensi yang relevan dan menarik. Salah satu orang pertama yang menyadari ini tahun lalu adalah**Edsger Dijkstra**(meskipun sulit untuk mengetahui sejarah pastinya [GR92]), dikenal antara lain untuk algoritma "jalur terpendek" yang terkenal dalam teori graf [D59], polemik awal pada pemrograman terstruktur berjudul "Pernyataan Goto Dianggap Berbahaya" [D68a] (judul yang bagus!), dan, dalam kasus yang akan kita pelajari di sini, pengenalan primitif sinkronisasi yang disebut**tiang sinyal** [D68b, D72]. Memang, Dijkstra dan rekan menemukan semaphore sebagai primitif tunggal untuk semua hal yang berhubungan dengan sinkronisasi; seperti yang akan Anda lihat, seseorang dapat menggunakan semaphore sebagai kunci dan variabel kondisi.

TDIACRUX: HOWTHAIKAMUSESEMAPHORES

Bagaimana kita bisa menggunakan semaphore alih-alih kunci dan variabel kondisi?
Apa definisi semafor? Apa itu semafor biner? Apakah mudah untuk membangun semaphore dari kunci dan variabel kondisi? Untuk membuat kunci dan variabel kondisi dari semaphore?

31.1 Semafor: Definisi

Semaphore adalah objek dengan nilai integer yang dapat kita manipulasi dengan dua rutinitas; dalam standar POSIX, rutinitas ini adalahtunggu sebentar() dansem posting()[1]. Karena nilai awal semaphore menentukan perilakunya, sebelum memanggil rutin lain untuk berinteraksi dengan semaphore, kita harus terlebih dahulu menginisialisasinya ke beberapa nilai, seperti yang dilakukan kode pada Gambar 31.1.

[1]Secara historis,tunggu sebentar()telah dipanggilP()oleh Dijkstra andsem posting()diteleponV().Bentuk-bentuk singkat ini berasal dari kata-kata Belanda; menariknya,*yang*Kata-kata Belanda yang mereka duga berasal dari telah berubah dari waktu ke waktu. Semula,P()berasal dari “passing” (melewati) dan V()dari "vrijgave" (rilis); nanti, tulis DijkstraP()berasal dari "prolaag", singkatan dari "probeer" (bahasa Belanda untuk "coba") dan "verlaag" ("penurunan"), danV()dari "verhoog" yang berarti "meningkatkan". Terkadang, orang memanggil mereka ke bawah dan ke atas. Gunakan versi bahasa Belanda untuk membuat teman Anda terkesan, atau membingungkan mereka, atau keduanya. Lihathttps://news.ycombinator.com/ item?id=8761539) untuk rincian.

```
1 # sertakan <semaphore.h>
2 sem_t s;
3 semi_init(&s, 0, 1);
```

Gambar 31.1:**Menginisialisasi Semaphore**

Pada gambar, kami mendeklarasikan semaphore s dan menginisialisasinya ke nilai 1 dengan meneruskan 1 sebagai argumen ketiga. Argumen kedua untuksemi init() akan disetel ke 0 di semua contoh yang akan kita lihat; ini menunjukkan bahwa semaphore dibagi antara utas dalam proses yang sama. Lihat halaman manual untuk perincian tentang penggunaan semaphore lainnya (yaitu, bagaimana mereka dapat digunakan untuk menyinkronkan akses di seluruh*berbeda*proses), yang membutuhkan nilai berbeda untuk argumen kedua itu.

Setelah semaphore diinisialisasi, kita dapat memanggil salah satu dari dua fungsi untuk berinteraksi dengannya,tunggu sebentar()atausem posting().Perilaku kedua fungsi tersebut terlihat pada Gambar 31.2.

Untuk saat ini, kami tidak peduli dengan pelaksanaan rutinitas ini, yang jelas membutuhkan kehati-hatian; dengan banyak utas yang memanggil tunggu sebentar() dansem posting(),ada kebutuhan yang jelas untuk mengelola bagian-bagian penting ini. Kami sekarang akan fokus pada bagaimana*menggunakan*primitif ini; nanti kita bisa mendiskusikan bagaimana mereka dibangun.

Kita harus membahas beberapa aspek penting dari antarmuka di sini. Pertama, kita bisa melihatnyatunggu sebentar()akan segera kembali (karena nilai semaphore adalah satu atau lebih tinggi ketika kita memanggiltunggu sebentar()),atau itu akan menyebabkan penelepon menunda eksekusi menunggu posting berikutnya. Tentu saja, beberapa utas panggilan dapat memanggiltunggu sebentar(),dan dengan demikian semua harus antri menunggu untuk dibangunkan.

Kedua, kita dapat melihat bahwasem posting()tidak menunggu beberapa kondisi tertentu untuk bertahan sepertitunggu sebentar()melakukan. Sebaliknya, itu hanya menambah nilai semaphore dan kemudian, jika ada utas yang menunggu untuk dibangunkan, membangunkan salah satu dari mereka.

Ketiga, nilai semaphore, ketika negatif, sama dengan jumlah thread yang menunggu [D68b]. Meskipun nilainya umumnya tidak terlihat oleh pengguna semaphore, invarian ini perlu diketahui dan mungkin dapat membantu Anda mengingat bagaimana semaphore berfungsi.

```
1 int sem_wait(sem_t *s) {
2         kurangi nilai semaphore s dengan satu tunggu jika nilai
3         semaphore s negatif
4 }
5
6 int sem_post(sem_t *s) {
7         menambah nilai semaphore s dengan satu
8         jika ada satu atau lebih utas yang menunggu, bangunkan satu
9 }
```

Gambar 31.2:**Semaphore: Definisi Menunggu Dan Posting**

```
1 sem_t m;
2 semi_init(&m, 0, X); // inisialisasi ke X; apa yang harus X?
3
4 sem_tunggu(&m);
5 // bagian kritis di sini
6 sem_post(&m);
```

Gambar 31.3:**Semaphore Biner (Yaitu, Kunci)**

Jangan khawatir (belum) tentang kemungkinan kondisi balapan di dalam semaphore; berasumsi bahwa tindakan yang mereka lakukan dilakukan secara atom. Kami akan segera menggunakan kunci dan variabel kondisi untuk melakukan hal ini.

31.2 Semaphore Biner (Kunci)

Kami sekarang siap untuk menggunakan semaphore. Penggunaan pertama kita adalah yang sudah kita kenal: menggunakan semaphore sebagai kunci. Lihat Gambar 31.3 untuk potongan kode; di dalamnya, Anda akan melihat bahwa kami hanya mengelilingi bagian penting yang menarik dengan asem tunggu()/sem posting() pasangan. Penting untuk membuat ini berfungsi, bagaimanapun, adalah nilai awal semaphoreM (diinisialisasi kexdalam gambar). Apa seharusnyaxmenjadi?

. . . (Coba pikirkan sebelum melanjutkan) ...

Melihat kembali definisi daritunggu sebentar()dansem posting()rutinitas di atas, kita dapat melihat bahwa nilai awal harus 1.

Untuk memperjelas ini, mari kita bayangkan sebuah skenario dengan dua utas. Utas pertama (Utas 0) memanggiltunggu sebentar();itu pertama-tama akan mengurangi nilai semaphore, mengubahnya menjadi 0. Kemudian, itu akan menunggu hanya jika nilainya *bukan*lebih besar dari atau sama dengan 0. Karena nilainya 0,tunggu sebentar() hanya akan kembali dan utas panggilan akan berlanjut; Thread 0 sekarang bebas masuk ke critical section. Jika tidak ada utas lain yang mencoba mendapatkan kunci saat Utas 0 berada di dalam bagian kritis, saat ia memanggilsem posting(),itu hanya akan mengembalikan nilai semaphore ke 1 (dan tidak membangunkan utas yang menunggu, karena tidak ada). Gambar 31.4 menunjukkan jejak skenario ini.

Kasus yang lebih menarik muncul ketika Thread 0 "memegang kunci" (yaitu, telah disebuttunggu sebentar()tapi belum dipanggilsem_posting()),dan utas lain (Utas 1) mencoba memasuki bagian kritis dengan memanggiltunggu sebentar(). Dalam hal ini, Thread 1 akan mengurangi nilai semaphore menjadi -1, dan

Nilai Semaphore	Benang 0	Benang 1
1		
1	panggilantunggu sebentar()	
0	tunggu sebentar()kembali	
0	(sekte kritik)	
0	panggilansem posting()	
1	sem posting()kembali	

Gambar 31.4:**Jejak Utas: Utas Tunggal Menggunakan Semaphore**

Val	Benang 0	Negara	Benang 1	Negara
1		Lari		Siap
1	panggilantunggu sebentar()	Lari		Siap
0	tunggu sebentar()kembali	Lari		Siap
0	(Sekte kritik dimulai)	Lari		Siap
0	*Mengganggu; Mengalihkan→T1*	Siap		Lari
0		Siap	panggilantunggu sebentar()	Lari
- 1		Siap	Desember	Lari
- 1		Siap	(sem<0)→tidur	Tidur
- 1		Lari	*Mengalihkan→T0*	Tidur
- 1	(sekte kritik berakhir)	Lari		Tidur
- 1	panggilansem posting()	Lari		Tidur
0	termasuk sem	Lari		Tidur
0	bangun (T1)	Lari		Siap
0	sem posting()kembali	Lari		Siap
0	*Mengganggu; Mengalihkan→T1*	Siap		Lari
0		Siap	tunggu sebentar()kembali	Lari
0		Siap	(sekte kritik)	Lari
0		Siap	panggilansem posting()	Lari
1		Siap	sem posting()kembali	Lari

Gambar 31.5:**Jejak Utas: Dua Utas Menggunakan Semaphore**

jadi tunggu (menempatkan dirinya untuk tidur dan melepaskan prosesor). Ketika Thread 0 berjalan lagi, pada akhirnya akan memanggilsem posting(),menambah nilai semaphore kembali ke nol, dan kemudian membangunkan utas yang menunggu (Utas 1), yang kemudian akan dapat memperoleh kunci untuk dirinya sendiri. Ketika Thread 1 selesai, itu akan menambah nilai semaphore lagi, mengembalikannya ke 1 lagi.

Gambar 31.5 menunjukkan jejak dari contoh ini. Selain tindakan utas, gambar menunjukkan**status penjadwal**dari setiap utas: Jalankan (utas sedang berjalan), Siap (yaitu, dapat dijalankan tetapi tidak berjalan), dan Tidur (utas diblokir). Perhatikan bahwa Thread 1 masuk ke status tidur ketika mencoba untuk mendapatkan kunci yang sudah dipegang; hanya ketika Thread 0 berjalan lagi, Thread 1 dapat dibangunkan dan berpotensi berjalan lagi.

Jika Anda ingin mengerjakan contoh Anda sendiri, coba skenario di mana banyak utas mengantri menunggu kunci. Berapa nilai semaphore selama pelacakan seperti itu?

Dengan demikian kita dapat menggunakan semaphore sebagai kunci. Karena kunci hanya memiliki dua status (dipegang dan tidak dipegang), kadang-kadang kita menyebut semaphore digunakan sebagai kunci a**semafor biner**. Perhatikan bahwa jika Anda menggunakan semaphore hanya dalam mode biner ini, itu bisa diimplementasikan dengan cara yang lebih sederhana daripada semaphore umum yang kami sajikan di sini.

31.3 Semaphore Untuk Memesan

Semaphore juga berguna untuk memesan acara dalam program bersamaan. Misalnya, utas mungkin ingin menunggu daftar menjadi tidak kosong,

```
1  sem_t s;
2
3  batal *anak(batal *arg) {
4          printf("anak\n");
5          sem_post(&s); // sinyal di sini: anak selesai return NULL;
6
7  }
8
9  int main(int argc, char *argv[]) {
10         semi_init(&s, 0, X); // apa yang seharusnya menjadi X?
11         printf("induk: mulai\n");
12         pthread_t c;
13         Pthread_create(&c, NULL, anak, NULL);
14         sem_tunggu(&s); // tunggu di sini untuk anak
15         printf("induk: akhir\n");
16         kembali 0;
17 }
```

Gambar 31.6:**Orang Tua Menunggu Anaknya**

sehingga dapat menghapus elemen darinya. Dalam pola penggunaan ini, kita sering menemukan satu utas*menunggu*untuk sesuatu terjadi, dan utas lain membuat sesuatu terjadi dan kemudian*memberi isyarat*bahwa itu telah terjadi, sehingga membangunkan utas yang menunggu. Dengan demikian, kami menggunakan semaphore sebagai**Memerintah** primitif (mirip dengan penggunaan**variabel kondisi**lebih awal).

Contoh sederhananya adalah sebagai berikut. Bayangkan sebuah utas membuat utas lain dan kemudian ingin menunggunya menyelesaikan eksekusinya (Gambar 31.6). Ketika program ini berjalan, kami ingin melihat yang berikut:

```
orang tua: mulai
anak
orang tua: akhir
```

Pertanyaannya, kemudian, adalah bagaimana menggunakan semaphore untuk mencapai efek ini; ternyata, jawabannya relatif mudah dipahami. Seperti yang Anda lihat dalam kode, orang tua cukup menelepontunggu sebentar()dan anaksem posting() untuk menunggu kondisi anak menyelesaikan eksekusinya menjadi benar. Namun, ini menimbulkan pertanyaan: berapa nilai awal semaphore ini?

(Sekali lagi, pikirkan di sini, daripada membaca di depan)

Jawabannya, tentu saja, nilai semaphore yang harus disetel adalah 0. Ada dua kasus yang perlu dipertimbangkan. Pertama, mari kita asumsikan bahwa orang tua membuat anak tetapi anak belum berjalan (yaitu, duduk dalam antrian siap tetapi tidak berjalan). Dalam hal ini (Gambar 31.7, halaman 6), orang tua akan memanggiltunggu sebentar()sebelum anak itu menelepon.sem posting();kami ingin orang tua menunggu anak itu berlari. Satu-satunya cara ini akan terjadi adalah jika nilai semaphore tidak lebih besar dari 0; maka, 0 adalah nilai awal. Induk berjalan, mengurangi semaphore (menjadi -1), lalu menunggu (tidur). Ketika anak akhirnya berlari, ia akan memanggilsem posting(),menambah nilai

Val	Induk	Negara	Anak	Negara
0	buat (Anak)	Lari	*(Anak ada, bisa lari)*	Siap
0	panggilantunggu sebentar()	Lari		Siap
- 1	Desember	Lari		Siap
- 1	(sem<0)→tidur	Tidur		Siap
- 1	*Mengalihkan→Anak*	Tidur	anak berlari	Lari
- 1		Tidur	panggilansem posting()	Lari
0		Tidur	inc sem	Lari
0		Siap	bangun (Orangtua)	Lari
0		Siap	sem posting()kembali	Lari
0		Siap	*Mengganggu→Induk*	Siap
0	tunggu sebentar()kembali	Lari		Siap

Gambar 31.7:**Jejak Utas: Orang Tua Menunggu Anak (Kasus 1)**

Val	Induk	Negara	Anak	Negara
0	buat (Anak)	Lari	*(Anak ada; bisa lari)*	Siap
0	*Mengganggu→Anak*	Siap	anak berlari	Lari
0		Siap	panggilansem posting()	Lari
1		Siap	inc sem	Lari
1		Siap	bangun (tidak ada)	Lari
1		Siap	sem posting()kembali	Lari
1	orang tua berlari	Lari	*Mengganggu→Induk*	Siap
1	panggilantunggu sebentar()	Lari		Siap
0	pengurangan sem	Lari		Siap
0	(sem≥0)→bangun	Lari		Siap
0	tunggu()kembali	Lari		Siap

Gambar 31.8:**Jejak Utas: Orang Tua Menunggu Anak (Kasus 2)**

semaphore ke 0, dan membangunkan induknya, yang kemudian akan kembali dari tunggu sebentar()dan menyelesaikan program.

Kasus kedua (Gambar 31.8) terjadi ketika anak berlari sampai selesai sebelum orang tua mendapat kesempatan untuk menelepontunggu sebentar().Dalam hal ini, anak pertama akan memanggilsem posting(),sehingga menambah nilai semaphore dari 0 menjadi 1. Ketika orang tua kemudian mendapat kesempatan untuk menjalankan, itu akan memanggiltunggu sebentar()dan temukan nilai semaphore menjadi 1; orang tua dengan demikian akan mengurangi nilai (menjadi 0) dan kembali daritunggu_sebentar() tanpa menunggu, juga mencapai efek yang diinginkan.

31.4 Masalah Produsen/Konsumen (Bounded Buffer)

Masalah berikutnya yang akan kita hadapi dalam bab ini dikenal sebagai **produsen/konsumen**masalah, atau terkadang sebagai**buffer terbatas**masalah [D72]. Masalah ini dijelaskan secara rinci pada bab sebelumnya tentang variabel kondisi; lihat di sana untuk detailnya.

ASAMPING: SETTINGTDIAVALUEHAIFSEBAGAIEMAFORE

Kami sekarang telah melihat dua contoh inisialisasi semaphore. pertama kasus, kami menetapkan nilai ke 1 untuk menggunakan semaphore sebagai kunci; di detik, ke 0, menggunakan semaphore untuk memesan. Jadi apa aturan umum untuk inisialisasi semaphore?

Satu cara sederhana untuk memikirkannya, berkat Perry Kivolowitz, adalah dengan pertimbangkan jumlah sumber daya yang ingin Anda berikan segera setelah inisialisasi. Dengan kunci, itu adalah 1, karena Anda bersedia untuk mengunci kunci (diberikan) segera setelah inisialisasi. Dengan kasus pemesanan, itu adalah 0, karena tidak ada yang bisa diberikan di awal; hanya ketika utas anak selesai adalah sumber daya dibuat, pada titik mana, nilainya bertambah menjadi 1. Coba garis pemikiran ini tentang masalah semaphore di masa depan, dan lihat apakah itu membantu.

Percobaan pertama

Upaya pertama kami untuk memecahkan masalah memperkenalkan dua semafor,kosong danpenuh,yang akan digunakan utas untuk menunjukkan kapan entri buffer telah dikosongkan atau diisi, masing-masing. Kode untuk rutinitas put and get ada di Gambar 31.9, dan upaya kami untuk memecahkan masalah produsen dan konsumen ada di Gambar 31.10 (halaman 8).

Dalam contoh ini, produsen pertama-tama menunggu buffer menjadi kosong untuk memasukkan data ke dalamnya, dan konsumen juga menunggu buffer terisi sebelum menggunakannya. Mari kita bayangkan duluMAKSIMAL = 1 (hanya ada satu buffer dalam array), dan lihat apakah ini berfungsi.

Bayangkan lagi ada dua utas, produsen dan konsumen. Mari kita periksa skenario tertentu pada satu CPU. Asumsikan konsumen akan menjalankan lebih dulu. Dengan demikian, konsumen akan menekan Jalur C1 pada Gambar 31.10, memanggilsem menunggu (& penuh).Karena full diinisialisasi ke nilai 0,

```
1  int penyangga[MAX];
2  int isi = 0;
3  int digunakan = 0;
4
5  void put(nilai int) {
6          buffer[isi] = nilai;                    //    Jalur F1
7          isi = (isi + 1) % MAX; // Garis F2
8  }
9
10 int dapatkan() {
11         int tmp = penyangga[gunakan];          // Jalur G1
12         gunakan = (gunakan + 1) % MAX;         // Jalur G2
13         kembali tmp;
14 }
```

Gambar 31.9:**Put Dan Dapatkan Rutinitas**

```
1  sem_t kosong;
2  sem_t penuh;
3
4  void *produser(void *arg) {
5          di aku;
6          for (i = 0; i < loop; i++) {
7                  sem_tunggu(&kosong);          // Baris P1
8                  menempatkan (i);              // Baris P2
9                  sem_post(&penuh);             // Baris P3
10         }
11 }
12
13 void *konsumen(void *arg) {
14         inttmp = 0;
15         sementara (tmp != -1) {
16                 sem_wait(&penuh);             // Baris C1
17                 tmp = dapatkan();             // Baris C2
18                 sem_post(&kosong);            // Baris C3
19                 printf("%d\n",tmp);
20         }
21 }
22
23 int main(int argc, char *argv[]) {
24         // ...
25         sem_init(&kosong, 0, MAX); // MAX kosong
26         sem_init(&penuh, 0, 0); // 0 penuh // ...
27
28 }
```

Gambar 31.10:**Menambahkan Kondisi Penuh Dan Kosong**

panggilan akan berkurangpenuh (ke -1), blokir konsumen, dan tunggu utas lain dipanggilsem posting()padapenuh,seperti yang diinginkan.

Asumsikan produser kemudian berjalan. Itu akan mengenai Jalur P1, sehingga memanggil tunggu (&kosong)rutin. Berbeda dengan konsumen, produsen akan terus melalui jalur ini, karenakosongdiinisialisasi ke nilai MAKSIMAL (dalam hal ini, 1). Dengan demikian,kosongakan dikurangi menjadi 0 dan produser akan memasukkan nilai data ke entri buffer pertama (Jalur P2). Produser kemudian akan melanjutkan ke P3 dan meneleponsem posting(&penuh),mengubah nilai penuhsemaphore dari -1 ke 0 dan membangunkan konsumen (misalnya, memindahkannya dari diblokir ke siap).

Dalam hal ini, salah satu dari dua hal bisa terjadi. Jika produser terus berjalan, ia akan berputar dan menekan Jalur P1 lagi. Namun, kali ini, itu akan memblokir, karenakosongnilai semaphore adalah 0. Jika produsen malah terganggu dan konsumen mulai berjalan, itu akan kembali dari sem menunggu(&penuh) (Baris C1), temukan bahwa buffer sudah penuh, dan konsumsilah. Dalam kedua kasus, kami mencapai perilaku yang diinginkan.

Anda dapat mencoba contoh yang sama ini dengan lebih banyak utas (misalnya, banyak produsen, dan banyak konsumen). Ini masih harus bekerja.

```
1  void *produser(void *arg) {
2      di aku;
3      for (i = 0; i < loop; i++) {
4          sem_wait(&mutex);          // Baris P0 (GARIS BARU) //
5          sem_tunggu(&kosong);       Baris P1
6          menempatkan (i);           // Baris P2
7          sem_post(&penuh);          // Baris P3
8          sem_post(&mutex);          // Baris P4 (GARIS BARU)
9      }
10 }
11
12 void *konsumen(void *arg) {
13     di aku;
14     for (i = 0; i < loop; i++) {
15         sem_wait(&mutex);          // Baris C0 (GARIS BARU) //
16         sem_wait(&penuh);          Baris C1
17         int tmp = dapatkan();      // Baris C2
18         sem_post(&kosong);         // Baris C3
19         sem_post(&mutex);          // Baris C4 (GARIS BARU)
20         printf("%d\n",tmp);
21     }
22 }
```

Gambar 31.11:**Menambahkan Mutual Exclusion (Salah)**

Sekarang mari kita bayangkan ituMAKSIMALlebih besar dari 1 (katakanlahMAKS = 10). Untuk contoh ini, mari kita asumsikan bahwa ada banyak produsen dan banyak konsumen. Kami sekarang memiliki masalah: kondisi balapan. Apakah Anda melihat di mana itu terjadi? (Luangkan waktu dan carilah) Jika Anda tidak dapat melihatnya, berikut petunjuknya: lihat lebih dekat padataruh()danDapatkan()kode.

Oke, mari kita pahami masalahnya. Bayangkan dua produsen (Pa dan Pb) keduanya memanggil ketaruh()pada waktu yang hampir bersamaan. Asumsikan produser Pa dijalankan terlebih dahulu, dan baru mulai mengisi entri buffer pertama (isi = 0di Jalur F1). Sebelum Pa mendapat kesempatan untuk meningkatkanmengisicounter ke 1, itu terputus. Producer Pb mulai berjalan, dan pada Line F1 ia juga memasukkan datanya ke dalam elemen buffer ke-0, yang berarti bahwa data lama di sana akan ditimpa! Tindakan ini adalah tidak-tidak; kami tidak ingin ada data dari produsen yang hilang.

Solusi: Menambahkan Mutual Exclusion

Seperti yang Anda lihat, apa yang kami lupakan di sini adalah*pengecualian bersama*. Pengisian buffer dan penambahan indeks ke dalam buffer adalah bagian penting, dan karenanya harus dijaga dengan hati-hati. Jadi mari kita gunakan semaphore biner teman kita dan tambahkan beberapa kunci. Gambar 31.11 menunjukkan upaya kami.

Sekarang kita telah menambahkan beberapa kunci di sekitar seluruh bagian kode put()/get(), seperti yang ditunjukkan olehGARIS BARUkomentar. Sepertinya itu ide yang tepat, tetapi juga tidak berhasil. Mengapa? Jalan buntu. Mengapa terjadi kebuntuan? Luangkan waktu sejenak untuk mempertimbangkannya; mencoba untuk menemukan kasus di mana kebuntuan muncul. Urutan langkah apa yang harus terjadi agar program menemui jalan buntu?

```
1  void *produser(void *arg) {
2      di aku;
3      for (i = 0; i < loop; i++) {
4          sem_tunggu(&kosong);          // Baris P1
5          sem_wait(&mutex);             // Baris P1.5 (MUTEX DI SINI) //
6          menempatkan (i);              Baris P2
7          sem_post(&mutex);             // Baris P2.5 (DAN DI SINI) //
8          sem_post(&penuh);             Baris P3
9      }
10 }
11
12 void *konsumen(void *arg) {
13     di aku;
14     for (i = 0; i < loop; i++) {
15         sem_wait(&penuh);             // Baris C1
16         sem_wait(&mutex);             // Baris C1.5 (MUTEX DI SINI) //
17         int tmp = dapatkan();         Baris C2
18         sem_post(&mutex);             // Baris C2.5 (DAN DI SINI) //
19         sem_post(&kosong);            Baris C3
20         printf("%d\n",tmp);
21     }
22 }
```

Gambar 31.12:**Menambahkan Mutual Exclusion (Benar)**

Menghindari Kebuntuan

Oke, setelah Anda mengetahuinya, inilah jawabannya. Bayangkan dua utas, satu produsen dan satu konsumen. Konsumen harus menjalankan terlebih dahulu. Ia memperoleh mutex (Jalur C0), dan kemudian memanggiltunggu sebentar()pada semafor penuh (Jalur C1); karena belum ada data, panggilan ini menyebabkan konsumen memblokir dan dengan demikian menghasilkan CPU; penting, meskipun, konsumen masih memegang kunci.

Seorang produser kemudian berjalan. Ia memiliki data untuk diproduksi dan jika dapat dijalankan, ia akan dapat membangunkan utas konsumen dan semuanya akan baik-baik saja. Sayangnya, hal pertama yang dilakukannya adalah menelepontunggu sebentar()pada semaphore mutex biner (Garis P0). Kunci sudah dipegang. Oleh karena itu, produsen sekarang juga terjebak menunggu.

Ada siklus sederhana di sini. Konsumen*memegang*mutex dan adalah *menunggu* bagi seseorang untuk memberi sinyal penuh. Produser bisa*sinyal*penuh tapi adalah *menunggu*untuk mutexnya. Jadi, produsen dan konsumen masing-masing terjebak menunggu satu sama lain: kebuntuan klasik.

Akhirnya, Solusi yang Bekerja

Untuk mengatasi masalah ini, kita hanya harus mengurangi ruang lingkup kunci. Gambar 31.12 (halaman 10) menunjukkan solusi yang benar. Seperti yang Anda lihat, kami cukup memindahkan akuisisi dan pelepasan mutex ke sekitar bagian kritis;

kode tunggu dan sinyal penuh dan kosong ditinggalkan di luar[2]. Hasilnya adalah buffer terbatas yang sederhana dan berfungsi, pola yang umum digunakan dalam program multithread. Pahami sekarang; menggunakannya nanti. Anda akan berterima kasih kepada kami selama bertahun-tahun yang akan datang. Atau setidaknya, Anda akan berterima kasih kepada kami ketika pertanyaan yang sama ditanyakan pada ujian akhir, atau saat wawancara kerja.

31.5 Kunci Pembaca-Penulis

Masalah klasik lainnya berasal dari keinginan untuk penguncian primitif yang lebih fleksibel yang mengakui bahwa akses struktur data yang berbeda mungkin memerlukan jenis penguncian yang berbeda. Misalnya, bayangkan sejumlah operasi daftar bersamaan, termasuk penyisipan dan pencarian sederhana. Sementara sisipan mengubah keadaan daftar (dan dengan demikian bagian kritis tradisional masuk akal), pencarian cukup *Baca* struktur data; selama kami dapat menjamin bahwa tidak ada penyisipan yang sedang berlangsung, kami dapat mengizinkan banyak pencarian untuk dilanjutkan secara bersamaan. Jenis kunci khusus yang sekarang akan kami kembangkan untuk mendukung jenis operasi ini dikenal sebagai **kunci pembaca-penulis** [CHP71]. Kode untuk kunci tersebut tersedia pada Gambar 31.13 (halaman 12).

Kodenya cukup sederhana. Jika beberapa utas ingin memperbarui struktur data yang bersangkutan, itu harus memanggil pasangan baru operasi sinkronisasi: rwlock memperoleh writelock(), untuk mendapatkan kunci tulis, dan rwlock rilis writelock(), untuk melepaskannya. Secara internal, ini hanya menggunakan kunci tulis semaphore untuk memastikan bahwa hanya satu penulis yang dapat memperoleh kunci dan dengan demikian masuk ke bagian kritis untuk memperbarui struktur data yang bersangkutan.

Lebih menarik adalah sepasang rutinitas untuk memperoleh dan melepaskan kunci baca. Saat memperoleh kunci baca, pembaca terlebih dahulu memperoleh lock dan kemudian menambah pembacavariabel untuk melacak berapa banyak pembaca saat ini di dalam struktur data. Langkah penting kemudian diambil dalam rwlock memperoleh readlock() terjadi ketika pembaca pertama memperoleh kunci; dalam hal ini, pembaca juga memperoleh kunci tulis dengan memanggil tunggu sebentar() pada kunci tulis semaphore, dan kemudian melepaskan kunci dengan menyebut sem posting().

Jadi, setelah pembaca memperoleh kunci baca, lebih banyak pembaca akan diizinkan untuk mendapatkan kunci baca juga; namun, setiap utas yang ingin mendapatkan kunci tulis harus menunggu sampai *semua* pembaca selesai; yang terakhir keluar dari panggilan bagian kritis sem posting() pada "writelock" dan dengan demikian memungkinkan penulis yang menunggu untuk mendapatkan kunci.

Pendekatan ini berhasil (seperti yang diinginkan), tetapi memiliki beberapa hal negatif, terutama dalam hal keadilan. Secara khusus, akan relatif mudah bagi pembaca untuk membuat penulis kelaparan. Ada solusi yang lebih canggih untuk masalah ini; mungkin Anda bisa memikirkan implementasi yang lebih baik? Petunjuk: pikirkan tentang apa yang perlu Anda lakukan untuk mencegah lebih banyak pembaca memasuki kunci begitu seorang penulis menunggu.

[2] Memang, mungkin lebih alami untuk menempatkan mutex memperoleh/melepaskan di dalam taruh() dan Dapatkan() fungsi untuk tujuan modularitas.

```
1  typedef struct _rwlock_t {
2      kunci sem_t;              // semaphore biner (kunci dasar)
3      kunci tulis sem_t; // izinkan SATU penulis/BANYAK pembaca int
4      pembaca; // #pembaca di bagian kritis } rwlock_t;
5
6
7  batal rwlock_init(rwlock_t *rw) {
8      rw->pembaca = 0;
9      sem_init(&rw->kunci, 0, 1); sem_init(&rw-
10     >writelock, 0, 1);
11 }
12
13 batal rwlock_acquire_readlock(rwlock_t *rw) {
14     sem_wait(&rw->kunci);
15     rw->pembaca++;
16     if (rw->readers == 1) // reader pertama mendapat writelock
17         sem_wait(&rw->writelock);
18     sem_post(&rw->kunci);
19 }
20
21 batal rwlock_release_readlock(rwlock_t *rw) {
22     sem_wait(&rw->kunci);
23     rw->pembaca--;
24     if (rw->readers == 0) // pembaca terakhir lepaskan
25         sem_post(&rw->writelock);
26     sem_post(&rw->kunci);
27 }
28
29 batal rwlock_acquire_writelock(rwlock_t *rw) {
30     sem_wait(&rw->writelock);
31 }
32
33 batal rwlock_release_writelock(rwlock_t *rw) {
34     sem_post(&rw->writelock);
35 }
```

Gambar 31.13:**Kunci Pembaca-Penulis Sederhana**

Akhirnya, perlu dicatat bahwa kunci pembaca-penulis harus digunakan dengan hati-hati. Mereka sering menambahkan lebih banyak overhead (terutama dengan implementasi yang lebih canggih), dan dengan demikian tidak mempercepat kinerja dibandingkan dengan hanya menggunakan primitif penguncian sederhana dan cepat [CB08]. Either way, mereka menunjukkan sekali lagi bagaimana kita dapat menggunakan semaphore dengan cara yang menarik dan berguna.

TAKU P: SIMPLEANDDUMBCSEBUAHBEBSELESAI(HSAYA AKAN'SLAW)
Anda tidak boleh meremehkan gagasan bahwa pendekatan sederhana dan bodoh bisa menjadi yang terbaik. Dengan penguncian, terkadang kunci putar sederhana berfungsi paling baik, karena mudah diterapkan dan cepat. Meskipun sesuatu seperti kunci pembaca/penulis terdengar keren, mereka rumit, dan rumit bisa berarti lambat. Jadi, selalu coba pendekatan yang sederhana dan bodoh terlebih dahulu.

Gagasan tentang kesederhanaan ini ditemukan di banyak tempat. Salah satu sumber awal adalah disertasi Mark Hill [H87], yang mempelajari bagaimana merancang cache untuk CPU. Hill menemukan bahwa cache sederhana yang dipetakan langsung bekerja lebih baik daripada desain asosiatif set yang mewah (salah satu alasannya adalah bahwa dalam cache, desain yang lebih sederhana memungkinkan pencarian yang lebih cepat). Saat Hill dengan ringkas meringkas karyanya: "Besar dan bodoh lebih baik." Dan dengan demikian kami menyebut saran serupa ini **Hukum Bukit**.

31.6 Para Filsuf Bersantap

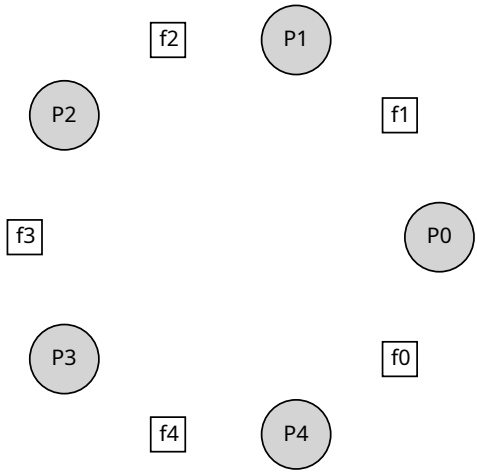
Salah satu masalah konkurensi paling terkenal yang diajukan, dan dipecahkan, oleh Dijkstra, dikenal sebagai**masalah filsuf makan**[D71]. Masalahnya terkenal karena menyenangkan dan agak menarik secara intelektual; namun, utilitas praktisnya rendah. Namun, ketenarannya memaksa dimasukkannya di sini; memang, Anda mungkin ditanya tentang itu pada beberapa wawancara, dan Anda akan sangat membenci profesor OS Anda jika Anda melewatkan pertanyaan itu dan tidak mendapatkan pekerjaan itu. Sebaliknya, jika Anda mendapatkan pekerjaan itu, jangan ragu untuk mengirimkan catatan bagus kepada profesor OS Anda, atau beberapa opsi saham.

Pengaturan dasar untuk masalahnya adalah ini (seperti yang ditunjukkan pada Gambar 31.14): asumsikan ada lima "filsuf" yang duduk di sekitar meja. Di antara setiap pasangan filsuf ada garpu tunggal (dan dengan demikian, total lima). Para filsuf masing-masing memiliki waktu di mana mereka berpikir, dan tidak membutuhkan garpu, dan waktu di mana mereka makan. Untuk makan, seorang filsuf membutuhkan dua garpu, baik yang di kiri dan yang di kanan. Perdebatan untuk garpu ini, dan masalah sinkronisasi yang terjadi, adalah apa yang membuat ini menjadi masalah yang kami pelajari dalam pemrograman konkuren.

Berikut adalah loop dasar dari setiap filsuf, dengan asumsi masing-masing memiliki pengenal utas yang unikPdari 0 hingga 4 (termasuk):

```
sementara (1) {
    memikirkan();
    get_forks(p);
    makan();
    put_forks(p);
}
```

Tantangan utamanya, kemudian, adalah menulis rutinitasambil garpu()dan taruh garpu() sedemikian rupa sehingga tidak ada kebuntuan, tidak ada filsuf yang kelaparan dan



Gambar 31.14:**Para Filsuf Bersantap**

tidak pernah bisa makan, dan konkurensinya tinggi (yaitu, sebanyak mungkin filsuf bisa makan pada waktu yang sama).

Mengikuti solusi Downey [D08], kami akan menggunakan beberapa fungsi pembantu untuk membawa kami menuju solusi. Mereka:

```
int kiri(int p) { kembali p; }
int kanan(int p) { kembali (p + 1) % 5; }
```

Ketika filsufPingin merujuk ke garpu di sebelah kiri mereka, mereka cukup memanggilkiri (hal).Demikian pula, garpu di sebelah kanan seorang filsufPdisebut dengan panggilankanan (p);operator modulo di dalamnya menangani satu kasus di mana filsuf terakhir (p=4)mencoba meraih garpu di sebelah kanannya, yaitu garpu 0.

Kami juga membutuhkan beberapa semaphore untuk menyelesaikan masalah ini. Mari kita asumsikan kita memiliki lima, satu untuk setiap garpu:sem t garpu[5].

Solusi Rusak

Kami mencoba solusi pertama kami untuk masalah ini. Asumsikan kita menginisialisasi setiap semaphore (dalamgarpuarray) ke nilai 1. Asumsikan juga bahwa setiap filsuf mengetahui nomornya sendiri (P).Dengan demikian kita dapat menulis ambil garpu()dantaruh garpu()rutin (Gambar 31.15, halaman 15).

Intuisi di balik solusi (rusak) ini adalah sebagai berikut. Untuk mendapatkan garpu, kami cukup mengambil "kunci" di masing-masing: pertama yang di sebelah kiri,

```
batal get_forks(int p) {
        sem_wait(&garpu[kiri(p)]);
        sem_wait(&garpu[kanan(p)]);
}

batalkan put_forks(int p) {
        sem_post(&garpu[kiri(p)]);
        sem_post(&garpu[kanan(p)]);
}
```

Gambar 31.15:**Ituambil garpu()Dantaruh garpu()Rutinitas**

```
batal get_forks(int p) {
        jika (p == 4) {
                sem_wait(&garpu[kanan(p)]);
                sem_wait(&garpu[kiri(p)]); } lain {

                sem_wait(&garpu[kiri(p)]);
                sem_wait(&garpu[kanan(p)]);
        }
}
```

Gambar 31.16:**Mematahkan Ketergantungan Dalamambil garpu()**

dan kemudian yang di sebelah kanan. Ketika kami selesai makan, kami melepaskannya. Sederhana, bukan? Sayangnya, dalam hal ini, sederhana berarti rusak. Bisakah Anda melihat masalah yang muncul? Pikirkan tentang itu.

Masalahnya adalah**jalan buntu**. Jika setiap filsuf kebetulan mengambil garpu di sebelah kiri mereka sebelum filsuf mana pun dapat mengambil garpu di kanannya, masing-masing akan terjebak memegang satu garpu dan menunggu yang lain, selamanya. Secara spesifik, filsuf 0 meraih garpu 0, filsuf 1 meraih garpu 1, filsuf 2 meraih garpu 2, filsuf 3 meraih garpu 3, dan filsuf 4 meraih garpu 4; semua garpu diperoleh, dan semua filsuf terjebak menunggu garpu yang dimiliki filsuf lain. Kami akan segera mempelajari kebuntuan secara lebih rinci; untuk saat ini, aman untuk mengatakan bahwa ini bukan solusi yang berfungsi.

Solusi: Mematahkan Ketergantungan

Cara paling sederhana untuk mengatasi masalah ini adalah dengan mengubah cara garpu diperoleh oleh setidaknya salah satu filsuf; memang, begitulah Dijkstra sendiri memecahkan masalah. Secara khusus, mari kita asumsikan bahwa filsuf 4 (yang bernomor tertinggi) mendapat garpu di a*berbeda*urutan dari yang lain (Gambar 31.16); itutaruh garpu()kode tetap sama.

Karena filsuf terakhir mencoba meraih tepat sebelum kiri, tidak ada situasi di mana setiap filsuf meraih satu garpu dan terjebak menunggu yang lain; siklus menunggu terputus. Pikirkan konsekuensi dari solusi ini, dan yakinkan diri Anda bahwa itu berhasil.

Ada masalah "terkenal" lainnya seperti ini, misalnya,**masalah perokok**atau **masalah tukang cukur tidur**. Kebanyakan dari mereka hanyalah alasan untuk memikirkan konkurensi; beberapa dari mereka memiliki nama yang menarik. Cari tahu jika Anda tertarik untuk belajar lebih banyak, atau hanya ingin lebih banyak berlatih berpikir secara bersamaan [D08].

31.7 Pelambatan Benang

Satu kasus penggunaan sederhana lainnya untuk semaphore kadang-kadang muncul, dan dengan demikian kami menyajikannya di sini. Masalah spesifiknya adalah ini: bagaimana seorang programmer dapat mencegah "terlalu banyak" utas melakukan sesuatu sekaligus dan menghambat sistem? Jawaban: tentukan ambang batas untuk "terlalu banyak", dan kemudian gunakan semaphore untuk membatasi jumlah utas yang secara bersamaan mengeksekusi potongan kode yang dimaksud. Kami menyebut pendekatan ini**pelambatan** [T99], dan menganggapnya sebagai bentuk**kontrol masuk**.

Mari kita pertimbangkan contoh yang lebih spesifik. Bayangkan Anda membuat ratusan utas untuk mengerjakan beberapa masalah secara paralel. Namun, di bagian kode tertentu, setiap utas memperoleh sejumlah besar memori untuk melakukan bagian dari perhitungan; sebut saja bagian kode ini sebagai *wilayah intensif memori*. Jika*semua*benang memasuki wilayah intensif memori pada saat yang sama, jumlah semua permintaan alokasi memori akan melebihi jumlah memori fisik pada mesin. Akibatnya, mesin akan mulai meronta-ronta (yaitu, menukar halaman ke dan dari disk), dan seluruh komputasi akan melambat hingga merangkak.

Semaphore sederhana dapat memecahkan masalah ini. Dengan menginisialisasi nilai semaphore ke jumlah maksimum utas yang ingin Anda masukkan ke wilayah intensif memori sekaligus, dan kemudian menempatkan tunggu sebentar() dansem posting()di sekitar wilayah, semaphore secara alami dapat membatasi jumlah utas yang pernah secara bersamaan berada di wilayah kode yang berbahaya.

31.8 Bagaimana Menerapkan Semaphore

Akhirnya, mari kita gunakan primitif sinkronisasi tingkat rendah, kunci dan variabel kondisi, untuk membangun versi semaphore kita sendiri yang disebut ... *(drum roll di sini)*...**Zemafor**. Tugas ini cukup mudah, seperti yang Anda lihat pada Gambar 31.17 (halaman 17).

Dalam kode di atas, kami hanya menggunakan satu kunci dan satu variabel kondisi, ditambah variabel status untuk melacak nilai semaphore. Pelajari sendiri kodenya sampai Anda benar-benar memahaminya. Lakukan!

Satu perbedaan halus antara Zemaphore dan semaphore murni seperti yang didefinisikan oleh Dijkstra adalah bahwa kita tidak mempertahankan invarian bahwa nilai semaphore, ketika negatif, mencerminkan jumlah thread yang menunggu; memang, nilainya tidak akan pernah lebih rendah dari nol. Perilaku ini lebih mudah diimplementasikan dan cocok dengan implementasi Linux saat ini.

```
typedef struct __Zem_t {
        nilai int;
        kondisi pthread_cond_t;
        kunci pthread_mutex_t;
} Zem_t;

// hanya satu thread yang dapat memanggil void
ini Zem_init(Zem_t *s, int value) {
        s->nilai = nilai;
        Cond_init(&s->cond);
        Mutex_init(&s->kunci);
}

batal Zem_wait(Zem_t *s) {
        Mutex_lock(&s->lock);
        sementara (s->nilai <= 0)
                Cond_wait(&s->cond, &s->lock); s->nilai--;

        Mutex_unlock(&s->lock);
}

batal Zem_post(Zem_t *s) {
        Mutex_lock(&s->lock);
        s->nilai++;
        Cond_signal(&s->cond);
        Mutex_unlock(&s->lock);
}
```

Gambar 31.17:**Menerapkan Zemaphores Dengan Kunci Dan CV**

Anehnya, membangun variabel kondisi dari semaphore adalah proposisi yang jauh lebih rumit. Beberapa programmer bersamaan yang sangat berpengalaman mencoba melakukan ini di lingkungan Windows, dan banyak bug yang berbeda terjadi [B04]. Cobalah sendiri, dan lihat apakah Anda dapat mengetahui mengapa membangun variabel kondisi dari semaphore lebih menantang daripada yang mungkin muncul.

31.9 Ringkasan

Semaphore adalah primitif yang kuat dan fleksibel untuk menulis program bersamaan. Beberapa programmer menggunakannya secara eksklusif, menghindari kunci dan variabel kondisi, karena kesederhanaan dan utilitasnya.

Dalam bab ini, kami hanya menyajikan beberapa masalah dan solusi klasik. Jika Anda tertarik untuk mengetahui lebih lanjut, ada banyak bahan lain yang bisa Anda referensikan. Satu yang bagus (dan referensi gratis) adalah buku Allen Downey tentang konkurensi dan pemrograman dengan semaphore [D08]. Buku ini memiliki banyak teka-teki yang dapat Anda kerjakan untuk meningkatkan pemahaman Anda-

TAKU P: BECAREFULWITUGENERALISASI

Teknik abstrak generalisasi dengan demikian dapat sangat berguna dalam desain sistem, di mana satu ide bagus dapat dibuat sedikit lebih luas dan dengan demikian memecahkan kelas masalah yang lebih besar. Namun, berhati-hatilah saat menggeneralisasi; seperti yang diperingatkan Lampson kepada kita, "Jangan menggeneralisasi; generalisasi umumnya salah" [L83].

Orang bisa melihat semaphore sebagai generalisasi dari kunci dan variabel kondisi; namun, apakah generalisasi seperti itu diperlukan? Dan, mengingat kesulitan mewujudkan variabel kondisi di atas semaphore, mungkin generalisasi ini tidak umum seperti yang Anda bayangkan.

ing dari kedua semaphore secara khusus dan konkurensi pada umumnya. Menjadi ahli konkurensi sejati membutuhkan usaha bertahun-tahun; melampaui apa yang Anda pelajari di kelas ini tidak diragukan lagi adalah kunci untuk menguasai topik semacam itu.

Referensi

[B04] "Menerapkan Variabel Kondisi dengan Semaphores" oleh Andrew Birrell. Desember 2004.*Bacaan yang menarik tentang betapa sulitnya menerapkan CV di atas semafor sebenarnya, dan kesalahan yang dibuat oleh penulis dan rekan kerja di sepanjang jalan. Sangat relevan karena grup tersebut telah melakukan banyak sekali pemrograman secara bersamaan; Birrell, misalnya, dikenal (antara lain) menulis berbagai panduan pemrograman utas.*

[CB08] "Konkurensi Dunia Nyata" oleh Bryan Cantrill, Jeff Bonwick. Antrian ACM. Jilid 6, No. 5. September 2008.*Artikel bagus oleh beberapa peretas kernel dari perusahaan yang sebelumnya dikenal sebagai Sun tentang masalah nyata yang dihadapi dalam kode bersamaan.*

[CHP71] "Kontrol Bersamaan dengan Pembaca dan Penulis" oleh PJ Courtois, F. Heymans, DL Parnas. Komunikasi ACM, 14:10, Oktober 1971.*Pengenalan masalah pembaca-penulis, dan solusi sederhana. Pekerjaan selanjutnya memperkenalkan solusi yang lebih kompleks, dilewati di sini karena, yah, mereka cukup kompleks.*

[D59] "Catatan tentang Dua Masalah Terkait dengan Graf" oleh EW Dijkstra. Numerische Mathematik 1, 269271, 1959. Tersedia:http://www-m3.ma.tum.de/twiki/pub/MN0506/ WebHome/dijkstra.pdf.*Bisakah Anda percaya orang-orang mengerjakan algoritme pada tahun 1959? Kami tidak bisa. Bahkan sebelum komputer menyenangkan untuk digunakan, orang-orang ini memiliki perasaan bahwa mereka akan mengubah dunia...*

[D68a] "Pernyataan Masuk Dianggap Berbahaya" oleh EW Dijkstra. CACM, volume 11(3), Maret 1968.http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD215.PDF.*Terkadang dianggap sebagai awal dari bidang rekayasa perangkat lunak.*

[D68b] "Struktur Sistem Multiprogramming" oleh EW Dijkstra. CACM, volume 11(5), 1968.*Salah satu makalah paling awal yang menunjukkan bahwa sistem bekerja dalam ilmu komputer adalah upaya intelektual yang menarik. Juga berpendapat kuat untuk modularitas dalam bentuk sistem berlapis.*

[D72] "Aliran Informasi Berbagi Penyangga Terbatas" oleh EW Dijkstra. Surat Pemrosesan Informasi 1, 1972. http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD329.PDF.*Apakah Dijkstra menciptakan segalanya? Tidak, tapi mungkin dekat. Dia pasti adalah orang pertama yang dengan jelas menuliskan apa masalahnya dalam kode bersamaan. Namun, praktisi dalam desain OS mengetahui banyak masalah yang dijelaskan oleh Dijkstra, jadi mungkin memberinya terlalu banyak pujian akan menjadi gambaran yang keliru.*

[D08] "Buku Kecil Semaphores" oleh AB Downey. Tersedia di situs berikut: http://greenteapress.com/semaphores/.*Buku yang bagus (dan gratis!) tentang semafor. Banyak masalah menyenangkan untuk dipecahkan, jika Anda menyukai hal semacam itu.*

[D71] "Pemesanan hierarkis proses sekuensial" oleh EW Dijkstra. Tersedia online di sini: http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD310.PDF.*Menyajikan banyak masalah konkurensi, termasuk Dining Philosophers. Halaman wikipedia tentang masalah ini juga berguna.*

[GR92] "Pemrosesan Transaksi: Konsep dan Teknik" oleh Jim Gray, Andreas Reuter. Morgan Kaufmann, September 1992.*Kutipan tepat yang kami temukan sangat lucu ditemukan di halaman 485, di bagian atas Bagian 8.8: "Multiprosesor pertama, sekitar tahun 1960, memiliki instruksi pengujian dan pengaturan ... mungkin para pelaksana OS mengerjakan algoritme yang sesuai, meskipun Dijkstra adalah umumnya dikreditkan dengan menciptakan semaphore bertahun-tahun kemudian. Oh, jepret!*

[H87] "Aspek Memori Cache dan Kinerja Penyangga Instruksi" oleh Mark D. Hill. Ph.D. Disertasi, UC Berkeley, 1987.*Karya disertasi Hill, bagi mereka yang terobsesi dengan caching di sistem awal. Sebuah contoh yang bagus dari disertasi kuantitatif.*

[L83] "Petunjuk untuk Desain Sistem Komputer" oleh Butler Lampson. Tinjauan Sistem Operasi ACM, 15:5, Oktober 1983. *Lampson, seorang peneliti sistem terkenal, senang menggunakan petunjuk dalam desain sistem komputer. Petunjuk adalah sesuatu yang sering benar tetapi bisa salah; dalam penggunaan ini, sebuah signal() memberi tahu utas tunggu bahwa itu mengubah kondisi yang sedang ditunggu oleh pelayan, tetapi tidak untuk percaya bahwa kondisinya akan berada dalam keadaan yang diinginkan ketika utas menunggu bangun. Dalam makalah ini tentang petunjuk untuk merancang sistem, salah satu petunjuk umum Lampson adalah bahwa Anda harus menggunakan petunjuk. Hal ini tidak membingungkan seperti kedengarannya.*

[T99] "Re: Orang kernel NT bermain dengan Linux" oleh Linus Torvalds. 27 Juni 1999. Tersedia: https://yarchive.net/comp/linux/semaphores.html.*Tanggapan dari Linus sendiri tentang kegunaan semaphore, termasuk kasus pelambatan yang kami sebutkan dalam teks. Seperti biasa, Linus sedikit menghina tapi cukup informatif.*

Pekerjaan Rumah (Kode)

Dalam pekerjaan rumah ini, kita akan menggunakan semaphore untuk memecahkan beberapa masalah konkurensi yang terkenal. Banyak dari ini diambil dari Downey's excellent "Buku Kecil Semaphore"[3], yang melakukan pekerjaan yang baik untuk mengumpulkan sejumlah masalah klasik serta memperkenalkan beberapa varian baru; pembaca yang tertarik harus memeriksa Buku Kecil untuk lebih menyenangkan.

Masing-masing pertanyaan berikut menyediakan kerangka kode; tugas anda adalah mengisi kode untuk membuatnya bekerja yang diberikan semaphore. Di Linux, Anda akan menggunakan semaphore asli; pada Mac (di mana tidak ada dukungan semaphore), Anda harus terlebih dahulu membangun implementasi (menggunakan kunci dan variabel kondisi, seperti yang dijelaskan dalam bab ini). Semoga berhasil!

pertanyaan

1. Masalah pertama hanyalah mengimplementasikan dan menguji solusi untuk**garpu/gabung masalah**, seperti yang dijelaskan dalam teks. Meskipun solusi ini dijelaskan dalam teks, tindakan mengetiknya sendiri bermanfaat; bahkan Bach akan menulis ulang Vivaldi, memungkinkan seseorang yang akan segera menjadi master untuk belajar dari master yang sudah ada. Melihatfork-join.cuntuk rincian. Tambahkan panggilantidur(1)kepada anak untuk memastikan itu berhasil.
2. Sekarang mari kita generalisasi ini sedikit dengan menyelidiki**masalah pertemuan**. Masalahnya adalah sebagai berikut: Anda memiliki dua utas, yang masing-masing akan memasuki titik pertemuan dalam kode. Tidak ada yang harus keluar dari bagian kode ini sebelum yang lain memasukinya. Pertimbangkan untuk menggunakan dua semaphore untuk tugas ini, dan lihatrendezvous.cuntuk rincian.
3. Sekarang selangkah lebih maju dengan menerapkan solusi umum untuk**sinkronisasi penghalang**. Asumsikan ada dua titik dalam potongan kode yang berurutan, yang disebutP_1danP_2. Menempatkan**penghalang**di antaraP_1danP_2menjamin bahwa semua utas akan dieksekusiP_1sebelum satu utas dieksekusiP_2. Tugas Anda: tulis kode untuk mengimplementasikan apenghalang()fungsi yang dapat digunakan dengan cara ini. Aman untuk berasumsi bahwa Anda tahuN (jumlah total utas dalam program yang sedang berjalan) dan itu semuanbenang akan mencoba memasuki penghalang. Sekali lagi, Anda mungkin harus menggunakan dua semaphore untuk mencapai solusi, dan beberapa bilangan bulat lainnya untuk menghitung sesuatu. Melihatpenghalang.cuntuk rincian.
4. Sekarang mari kita selesaikan**masalah pembaca-penulis**, juga seperti yang dijelaskan dalam teks. Dalam pengambilan pertama ini, jangan khawatir tentang kelaparan. Lihat kodenya dipembaca-penulis.c untuk rincian. Menambahkantidur()panggilan ke kode Anda untuk menunjukkannya berfungsi seperti yang Anda harapkan. Bisakah Anda menunjukkan keberadaan masalah kelaparan?
5. Mari kita lihat kembali masalah pembaca-penulis, tapi kali ini, khawatir kelaparan. Bagaimana Anda bisa memastikan bahwa semua pembaca dan penulis pada akhirnya membuat kemajuan? Lihatpembaca-penulis-nostarve.cuntuk rincian.
6. Gunakan semaphore untuk membangun a**mutex tanpa kelaparan**, di mana setiap utas yang mencoba memperoleh mutex pada akhirnya akan mendapatkannya. Lihat kodenya dimutex-nostarve.c untuk informasi lebih lanjut.
7. Menyukai masalah ini? Lihat teks gratis Downey untuk lebih banyak seperti mereka. Dan jangan lupa, bersenang-senanglah! Tapi, Anda selalu melakukannya saat menulis kode, bukan?

[3]Tersedia:http://greenteapress.com/semaphores/downey08semaphores.pdf.