YAZICI BENZETİM PROBLEMİ

Burada anlatılanlar yazacağımız simülasyon etrafında zuhur edecektir. Bu kullanıcılara ve yazıcılara sahip olan bir sistemdir. Burada nusers kadar kullanıcı ve nprinters kadar yazıcı vardır. Buradaki tüm yazıcıların özdeş olduğunu kabul ediyoruz. Böylece bir kullanıcı çıktı almak istediğinde hangi yazıcının çalıştığının bir önemi yoktur.
Şimdi simülasyonda bir kullanıcının çıktı almak istediğini düşünelim. Bu durumda çıktı alma işi sunulur. Eğer herhangi bir yazıcı müsait ise istenen işlem gerçekleştirilir (bir sayfa 4 saniyede yazılır). Eğer hiçbir yazıcı müsait değilse, herhangi bir tanesi boşalana kadar çıktı alma isteği bekletilir.

Bizim yazıcı kuyruğumuz sabit bir genişliktedir. Eğer kuyruk tam doluysa kullanıcı kuyruk boşalana kadar yazdırma işini sunmak için beklemek zorundadır.

Açıkça bellidir ki bu simülasyonu oluşturmak için iş parçacıklarını(thread) kullanmamız gerekmektedir. Her kullanıcı kendi iş parçacığına sahip olacaktır. Aynı zamanda her yazıcı da kendi iş parçacığına sahiptir. İş parçacıkları paylaşımlı hafıza üzerinden haberleşirler.

printqsim

Bu program özel bir formata sahiptir. İş parçacıkları laboratuarda paylaşılacaktır. **printqsim.h** başlık dosyasıdır. Bazı data yapılarını ve alt rutin prototipleri tanımlar. Laboratuar çalışmasını yapmak için bu alt rutinleri yazmak zorundasınız. **printqsim.c** ise sürücü programıdır. Bu, iş parçacığını düzenleyen **main()** rutinini tanımlar. Tanımlayacağınız alt rutinlerle birlikte sürücü programı problemi çözecektir.

Başlık veya sürücü dosyasını değiştiremezsiniz. Ama bunun yerine başlık dosyasındaki alt rutinleri tanımlayan bir C dosyası temin edebilirsiniz. Bu C dosyası sürücü programı ile birlikte derlendiğinde sonuç programı problemi çözecektir.

Bu durumda yapmamız gereken,

printqsim.c ile birlikte **initialize_v(), submit_job(), get_print_job()** dosyalarını tanımlamaktır. Programımız kullanıcı/yazıcı simülasyonunu düzgün bir şekilde gerçekleştirir.

printqsim.c sürücü programını inceleyelim. 6 nokta vardır:

1. **nusers**: kullanıcı sayısı
2. **nprinters**: yazıcı sayısı
3. **arrtime**: ortalama zaman. Kullanıcıların istekte bulundukları yazdırma işleri arasındaki zaman
4. **maxpages**: yazdırma işinin maksimum boyutu (sayfa olarak)
5. **bufsize**: yazıcı kuyruk boyutu
6. **nevents**: her bir kullanıcının yaptığı yazdırma işlerinin sayısı

main() fonksiyonu/metodu **Spq** yapısını kurar. Bu fonksiyon/metot **printqsim.h** içinde tanımlanmıştır. Her kullanıcı ve yazıcı iş parçacığı, argümanı olarak bu yapıların birisine ait bir işaretçi alacaktır ve iş parçacığını ihtiyaç duyduğu bütün bilgi bu yapı içinde olacaktır. Bu şekilde global değişken kullanımına ihtiyaç duymayacağız. Her komut satırı argümanı **Spq** yapısı içinde bir alana sahiptir, ek olarak aşağıdaki ekstra alanlar da mevcuttur:

- **id:** Kullanıcının/yazıcının kimliği
- **starttime:** program başladığındaki **time(0)** değeri. Bu, program çalışırken bilgiyi yazdırmada kullanışlıdır.
- **v:** Bu, kodunuzun içinde tanımlamak için alacağınız bir **(void *)** 'dir**. initialize_v()** içinde başlatacaksınız. Her **Spq** yapısının **v** alanı aynı değere işaret edecektir. Bunun nasıl bir bit içinde kullanılacağını göreceksiniz.

(içinde **initialize_v()**'ye bir çağrısı olan) bir **Spq** ayarlamasını yaptıktan sonra, **main()** iç parçacığı bir rastgele sayı üreticisi kurar ve sonra **nusers** kullanıcı ve **nprinters** yazıcı iş parçacıklarını oluşturur. Her iş parçacığı kendi Spq yapısını argüman olarak ele alır. Her Spq yapısı içinde farklı olan tek şey id'lerdir. Diğer her şey (v'ye işaret eden işaretçi de dahil) aynıdır. Sonunda, ana iş parçacığı sadece kullanıcı ve yazıcı iş parçacıklarını bırakarak çıkar.

Kullanıcı iş parçacıkları

Her kullanıcı iş parçacığı aynı şeyi yapar. **Nevents** yinelemeleri için tekrar yapar. He tekrarda, rastgele bir zaman periyodu (1i le **arrtime*2** arasında – bu arrtime bekleme zamanının ortalamasını verir) kadar uyur ve sonra bir yazdırma işi gönderir. Bu iş üç alana sahip (kullanıcının id'si, bir iş id'si (i) ve 1 ile **maxpages** arasında rastgele bir sayfa sayısı) bir **Job** yapısı ile temsil edilir. Sonra iş **submit_job** ile gönderilir.

nevents işini gönderdikten sonra, kullanıcı iş parçacığı çıkar. Kullanıcı iş parçacığı uyurken ve bir iş gönderirken çıktı alır.

Yazıcı iş parçacıkları

Her yazıcı iş parçacığı aynı şeyi yapar. Daima, önce **get_print_job()**'u kullanarak işi alır ve sonra o işi yazdırır şeklinde yineleme yapar. Her sayfa için 4 saniye uyuyarak yazdırma işini simule eder. Yadırdıktan sonra işlemi tekrarlar. Yazıcı iş parçacığı bir iş isterken ve bir işi yazdırırken çıktı alır.

Hayali bir çözüm

Geriye sadece **initialize_v(), submit_job()** ve **get_print_job()'**u yazmak kaldı. Yeniden tekrarlamak için bu bir laboratuar yerine işiniz bu üç alt programı/fonksiyonu/metodu yazmak olacaktı. Böylece onlar da printqsim.h ve printqsim.c ile çalışırlardı. **Printqsim.h** ve **printqsim.c** dosyalarını değiştirmeye izniniz olmazdı.

Şimdi **psl.c** dosyasına bakın. Bu, probleme bir çözümdür. Çalışan bir çözüm değil fakat derleyip çalıştıracağınız bir tanesi. Yaptığı **s->v**'yi **NULL** olarak ayarlamak, yazdırma işlerini gönderildiklerinde yoksaymak ve yazıcı iş parçacıklarını çıkmaları için zorlamaktır.

Çalıştırmayı deneyin:

```
UNIX> ps1 5 3 5 5 5 3
  0: user  0/000: Sleeping for  6 seconds
  0: user  1/000: Sleeping for  7 seconds
  0: user  2/000: Sleeping for  6 seconds
  0: user  3/000: Sleeping for  1 seconds
  0: user  4/000: Sleeping for 10 seconds
  0: prnt  0/000: ready to print
  0: prnt  0/000: Done
  0: prnt  1/000: ready to print
  0: prnt  1/000: Done
  0: prnt  2/000: ready to print
  0: prnt  2/000: Done
  1: user  3/000: Submitting a job with size 4
  1: user  3/001: Sleeping for  7 seconds
  6: user  2/000: Submitting a job with size 5
  6: user  2/001: Sleeping for  8 seconds
  6: user  0/000: Submitting a job with size 2
  6: user  0/001: Sleeping for  8 seconds
  7: user  1/000: Submitting a job with size 5
  7: user  1/001: Sleeping for  6 seconds
  8: user  3/001: Submitting a job with size 2
  8: user  3/002: Sleeping for  4 seconds
 10: user  4/000: Submitting a job with size 5
 10: user  4/001: Sleeping for  8 seconds
 12: user  3/002: Submitting a job with size 3
 12: user  3/003: Done
 13: user  1/001: Submitting a job with size 3
 13: user  1/002: Sleeping for  5 seconds
 14: user  0/001: Submitting a job with size 1
 14: user  0/002: Sleeping for 10 seconds
 14: user  2/001: Submitting a job with size 5
 14: user  2/002: Sleeping for  9 seconds
 18: user  4/001: Submitting a job with size 5
 18: user  4/002: Sleeping for  6 seconds
 18: user  1/002: Submitting a job with size 3
 18: user  1/003: Done
 23: user  2/002: Submitting a job with size 4
 23: user  2/003: Done
 24: user  0/002: Submitting a job with size 2
```

```
  24: user  0/003: Done
  24: user  4/002: Submitting a job with size 4
  24: user  4/003: Done
UNIX>
```

Bu, 5 kullanıcı 3 yazıcı yazdırma işleri arasında ortalama 5 saniye beklemeli, azami sayfa sayısı (max page) 5, yazdırma kuyruk boyutu 5 ve kullanıcı başına 3 yazdırma işi olan bir simülasyon oluşturdu.

Simülasyonun çalıştığını ama doğru çalışmadığını fark edeceksiniz. Neden? Önce, yazıcılar hiçbir şey yazmadılar. Dahası, beşten fazla yazdırma işi gönderildi ve görünüşte kuyruğa alındı ve sonraki yazdırma işleri hala gönderilmeye izinliydiler.

Bu aptal bir örnek gibi görünebilir fakat önemli bir şeyi tasvir ediyor – probleme ait çözümler derlenip çalıştırılabilir fakat onların çıkışlarını doğruluk için kontrol etmelisiniz. İş parçacığı laboratuarlarınız için, böyle doğru olmayan fakat size bir başlangıç noktası veren bunun gibi "çözümler" sunacağım.

Yazılan kod 5 kullanıcılı, 3 yazıcılı, yazdırma işleri arası ortalama 5 saniye, maksimum sayfa boyutu 5, yazdırma kuyruğu boyutu 5 ve her kullanıcı için 3 yazıcı işi olan bir simülasyon oluşturmuştur.

Simülasyonun çalıştığını gözleyeceksiniz ama bu gözlem doğru bir gözlem olmayacak. Yani yazıcılar yeni başlayanlar için asla çıktı vermedi. Ayrıca 5 ten fazla yazdırma işi gönderildi, kuyruğa alındı ve sonraki yazdırma işlerinin gönderilmesine izin verildi. Bu bir (bonehead) örneği gibi görünebilir ama önemli bir şey göstermektedir. Bir soruna çözüm üretmek için derleyip çalıştırabilirsiniz fakat yapılan bu işlemin doğruluğu için çıkışı kontrol etmelisiniz. Size uygulama laboratuarınız için çözümler sunmam yanlış olacaktır fakat en azından başlangıç için yararlı olacaktır.

Gerçek Bir çözüm üzerinden başlamak

Problemi çözmek için nasıl başlanacağı oldukça açıktır. **V** pointer'ınızda yazdırma işlevinin sırasını ayarlamamız gerekmektedir. Bu sıra **bufsize** olacaktır. Kullanıcı bir iş gönderdiğinde eğer kuyrukta buffersize'dan daha az yer öğe ise istenilen iş oraya konulacaktır. Aksi halde işlerden birini çıkartmak için bir yazıcıyı beklemek zorundasınız. Çünkü ara belleğe erişen birden çok iş parçacığı olduğundan, ara belleği bir mutex ile korumak zorundasınız. Yukarıdakilerin hepsi **ps2.c** programında yapıldı.

ps2.c programında ilk olarak buffer struct (tampon bellek yapısı) tanımlanır. Bu struct, kuyruk durumunu tanımlayan çevrimsel bir kuyruk gibi bir dizi kullanır. Aynı zamanda birde mutex'e sahiptir. **İnitialize_v()** metodunda buffer tahsis edilir ve v buffer olarak ayarlanır. Ayrıca eğer boş yer varsa, **submit_jub** metodu işi buffer'a ekler. Eğer boş yer yoksa kullanıcı thread'i (iş parçacığı) çıkar.

Bu örnek adım adım bir programlama örneğidir. Sizde devam etmeden önce çalışıyor olduğundan emin olunuz. Önceden olduğu gibi aynı parametrelerle bunu çağırdığımız zaman 5 işin gönderildiğini görürüz ve sonra tüm kullanıcılar çıkış yapar.

```
UNIX> ps2 5 3 5 5 5 3
  0: user  0/000: Sleeping for 10 seconds
  0: user  1/000: Sleeping for  5 seconds
  0: user  2/000: Sleeping for  8 seconds
  0: user  3/000: Sleeping for  3 seconds
  0: user  4/000: Sleeping for  6 seconds
  0: prnt  0/000: ready to print
  0: prnt  0/000: Done
  0: prnt  1/000: ready to print
  0: prnt  1/000: Done
  0: prnt  2/000: ready to print
  0: prnt  2/000: Done
  3: user  3/000: Submitting a job with size 2
  3: user  3/001: Sleeping for  1 seconds
  4: user  3/001: Submitting a job with size 2
  4: user  3/002: Sleeping for  6 seconds
  5: user  1/000: Submitting a job with size 5
  5: user  1/001: Sleeping for  6 seconds
  6: user  4/000: Submitting a job with size 2
  6: user  4/001: Sleeping for  2 seconds
  8: user  2/000: Submitting a job with size 2
  8: user  2/001: Sleeping for  6 seconds
  8: user  4/001: Submitting a job with size 3
  8: user  4 -- the queue is full -- exiting
 10: user  3/002: Submitting a job with size 3
 10: user  3 -- the queue is full -- exiting
 10: user  0/000: Submitting a job with size 5
 10: user  0 -- the queue is full -- exiting
 11: user  1/001: Submitting a job with size 3
 11: user  1 -- the queue is full -- exiting
 14: user  2/001: Submitting a job with size 5
 14: user  2 -- the queue is full -- exiting
UNIX>
```

Bir yarı-çalışan çözüm

Şimdi aklımıza gelen soru şu: kuyruk dolu olduğunda ne yapmalıyız. Ayrıca **get_print_job()** metodunu yazmaya başladığımızda, kuyruk boş olduğunda ve yazılacak iş yok ise ne yaparız. **ps3.c** programı tek bir çözümü destekler. **ps3.c** iyi bir çözüm değildir fakat çalışmaktadır. **submit_job()** metodu çağırıldığında ve kuyruk dolu ise mutex'e izin verilir ve **sleep(1)**

metodu çağırılır. Sonra kuyruk tekrar kontrol edilir. Bu yolla bir yazıcı iş parçacığı bu zaman zarfında **get_print_job()** metodunu çağırırsa kuyruktaki iş askıya alınabilir ve sonra kullanıcının işi kuyruğa gönderilebilir. Benzer şeklide kuyruk boş olduğunda ve bir yazıcı **get_print_job()** metodunu çağırırsa yazıcı birkaç saniye uykuya geçer ve tekrar kontrol eder.

Aşağıda çalışan bir kod parçacığı verilmiştir, bunu deneyin ve gözlemleyin.

```
UNIX> ps3 5 3 5 5 5 3
  0: user  0/000: Sleeping for 10 seconds
  0: user  1/000: Sleeping for  1 seconds
  0: user  2/000: Sleeping for  4 seconds
  0: user  3/000: Sleeping for  1 seconds
  0: user  4/000: Sleeping for 10 seconds
  0: prnt  0/000: ready to print
  0: prnt  0 sleeping because the queue is empty
  0: prnt  1/000: ready to print
  0: prnt  1 sleeping because the queue is empty
  0: prnt  2/000: ready to print
  0: prnt  2 sleeping because the queue is empty
  1: user  1/000: Submitting a job with size 3
  1: user  1/001: Sleeping for  7 seconds
  1: user  3/000: Submitting a job with size 4
  1: user  3/001: Sleeping for  4 seconds
  1: prnt  0/000: Printing job   0 from user  1 size   3
  1: prnt  1/000: Printing job   0 from user  3 size   4
  1: prnt  2 sleeping because the queue is empty
  2: prnt  2 sleeping because the queue is empty
  3: prnt  2 sleeping because the queue is empty
  4: user  2/000: Submitting a job with size 4
  4: user  2/001: Sleeping for 10 seconds
  4: prnt  2/000: Printing job   0 from user  2 size   4
  5: user  3/001: Submitting a job with size 1
  5: user  3/002: Sleeping for  2 seconds
  7: user  3/002: Submitting a job with size 2
  7: user  3/003: Done
  8: user  1/001: Submitting a job with size 5
  8: user  1/002: Sleeping for  4 seconds
 10: user  4/000: Submitting a job with size 3
 10: user  4/001: Sleeping for  9 seconds
 10: user  0/000: Submitting a job with size 5
 10: user  0/001: Sleeping for  5 seconds
 12: user  1/002: Submitting a job with size 3
 12: user  1 sleeping because the queue is full
 13: prnt  0/001: ready to print
```

```
  13: prnt  0/001: Printing job   1 from user  3 size   1
  13: user  1/003: Done
  14: user  2/001: Submitting a job with size 1
  14: user  2 sleeping because the queue is full
  15: user  0/001: Submitting a job with size 1
  15: user  0 sleeping because the queue is full
  15: user  2 sleeping because the queue is full
  16: user  2 sleeping because the queue is full
  16: user  0 sleeping because the queue is full
  17: prnt  1/001: ready to print
  17: prnt  1/001: Printing job   2 from user  3 size   2
  17: user  0/002: Sleeping for  3 seconds
  17: prnt  0/002: ready to print
...
  60: prnt  1 sleeping because the queue is empty
  60: prnt  2/004: ready to print
  60: prnt  2 sleeping because the queue is empty
  60: prnt  0 sleeping because the queue is empty
  61: prnt  2 sleeping because the queue is empty
  61: prnt  0 sleeping because the queue is empty
  61: prnt  1 sleeping because the queue is empty
  62: prnt  0 sleeping because the queue is empty
  62: prnt  1 sleeping because the queue is empty
  62: prnt  2 sleeping because the queue is empty
  63: prnt  0 sleeping because the queue is empty
  63: prnt  1 sleeping because the queue is empty
  63: prnt  2 sleeping because the queue is empty
  64: prnt  1 sleeping because the queue is empty
^CUNIX>
```

Yapılacak işler bittiğinde öncelikle queue (kuyruk) kontrol edilir, bir iş (jop) olmaması durumunda sistem (sleeping) moda geçer ve sonunda **cntl-c** ile program sonlandırılır. Bu çalışan bir çözümdür ancak yeterli değildir. Bu teknikte kuyruk periyodik olarak kontrol edilir buna "polling" denir. Kuyruğa bir iş (jop) gelir gelmez sistem hazır hale geçer ve yazma işlemi yapılır. Kullanıcının gönderdiği iş tamamlanır tamamlanmaz diğer bir iş gelene kadar yazıcı iş parçacığı kuyruğu boşaltılır. Kısacası,"polling" yazıcı iş parçacığı için iş gören bir teknik olmasına rağmen mükemmel değildir. Çalışmalarınızda polling metotunu görmeyi tercih etmeyiz.

Monitörler ve Durum Değişkenleri

"Monitörler" ve "durum değişkenleri" senkronizasyon için uygun araçlardır. Monitör ve durum değişkenlerinin iş parçacığı(thread) dilinin bir parçası mı yoksa iş parçacığı(thread) kütüphanesinin bir parçası mı olduğu bir tartışma konusudur. Kitabın 6 bölümünde bu değişkenlerin iş parçacığı (thread) dilinin bir parçası olduğu konusu üzerinde durulmuştur, bu

bölümde ise iş parçacığı(thread) kütüphanesinin bir parçası olmaları durumundan bahsedilecektir.“Monitor” iş parçacıkları (thread) “giriş” ve “çıkış” olan bir veri yapısıdır. Bir zaman aralığında monitör de sadece bir iş parçacığı olur. Tıpkı “mutex “ ve pthreadler gibi. Aslında kullanılan “mutex”dir, “monitör” diye bir şey yoktur. Durum değişkenleri *(Condition variables*) monitörle mutlaka bağlantı kurmalıdır. Durum değişkenlerinin etkilendiği üç işlem durumu vardır.

- **pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *mon)**

Kod parçacığında anlatılan, başka bir iş parçacığı engellenmeyi kaldırana kadar, mutex bırakılır ve engellenir. Bu işlem otomatik olarak yapılır. **pthread_cond_wait()** döndüğünde, yani sistem uyandığında, mutex yeniden alınır.

- **pthread_cond_signal(pthread_cond_t *c)**

Bu seçimde durum değişkenlerinde bir veya birden fazla iş parçacığı engellenmiştir ve onların engelleri kaldırılır. Durum değişkenlerinde engellenmiş iş parçacığı yoksa, **pthread_cond_signal()** çalışmaz. Birden fazla engellenmiş iş parçacığı varsa bu kod parçacığı iş parçacıklarının engelinin kaldırılacağının garantisini vermez. **pthread_cond_signal()** çağrıldığında pthreads kütüphanesi mutex’e gereksinim duymaz. Bırakılmış mutexleri kilitlemede kullanıcı iyi bir yöntemdir.

- **pthread_cond_broadcast(pthread_cond_t *c)**

Durum değişkenlerinde engellenmiş bütün iş parçacıklarının engellenmesi kaldırılır.

Sizden istediğim, monitör ve durum değişkenlerinin geri dönüş değerlerini test etmeniz.

Eğer **pthread_cond_signal()** ya da **pthread_cond_broadcast()**, çağrılırsa, mutex engellenmiş olmalıdır. Ayrıca, **pthread_cond_wait()** çağrıldığında, engellenmesi kaldırılmış iş parçacıkları mutexsi engeller. Bu ilk bakışta bir çelişki gibi görünüyor ancak hatırlanması gereken, iş parçacığı engelliyken ,bekleyen iş parçacığı mutex’in engellenmesini kaldırır. Engellenmesi kaldırılmış olduğunda ise, iş parçacığı mutex tekrar dönmeden onu **pthread_cond_wait** dan tekrar engeller.

İş parçacığı (thread) sisteminin durum değişkenlerinde uygulamasında bir kaç seçim alanı bulunmaktadır.

1. Blok olmayan iş parçacıklar **pthread_cond_signal()/pthread_cond_broadcast()** çalıştırmak için mutex kilidine kadar beklemek zorundadır. Yani, yalnızca blok olmayan durum değişkeni yerine mutex bloklaması yapılır.
2. Blok olmayan iş parçacığı mutex üzerindeki bloke gider mutex ve iş parçacığı **pthread_cond_signal()/pthread_cond_broadcast()** komut ile otomatik olarak kilitler. Mutex serbet kalır iş parçacığı takibinde bu komut

pthread_cond_signal()/pthread_cond_broadcast() çağrısı ile sürekli çalışır ve yeniden girilir.

Her iki yaklaşım için de görüşler mevcuttur. Pthread'da önceki yaklaşımı alır. Buradaki kişisel yorumum, sizin çalışma yaklaşımınız kendi belirleyeceğiniz programdır. Diğer yolda **pthread_cond_signal()** veya **pthread_cond_broadcast** komutları çağrıldıktan hemen sonra mutex'in kilitli olduğundan emin olmaktır. Benim kod her zaman bunu yapar.
Bunu daha ileri bir tartışma için kitap (Bölüm 6) okuyun.

Durum değişkenlerini kullanma

Şuanda bizim programımıza ilave edilen durum koşulları bellidir. Kuyruğun tamamen dolu olduğu ve kuyruğun tamamen boş olduğu iki tane durum değişkenine ihtiyacımız vardır. Kuyruk tamamen dolu iken **submit_job()** içindeki **pthread_cond_wait()** komutu çağırılır ve **get_print_job()** içindeki **pthread_cond_signal()** komutu çağırıldığında yazıcı iş parçacığı tamamen dolu olan kuyruktan bir işi siler.

Benzer şekilde kuyruk boş iken **get_print_job()** içindeki **pthread_cond_wait()** çağrıldığında ve **submit_job()** içindeki **pthread_cond_signal()** çağırıldığında bir kullanıcı iş parçacığı boş olan kuyruğa bir ekler.

Dikkat edilmeli ki **submit_job()** ve **get_print_job()** her iki komut döngüde kullanılır çünkü **pthread_cond_wait()** geri döndürüldüğünde mutex'i elde etme ve iş parçacığının bloke olmadan beklemesi arasındaki zamanda diliminde kuyruk boş veya dolu olabilir. Bu yüzden tekrar bekleme durumuna geçebilir. **ps4.c** çalıştırıldığında her şey iyi bir şekilde görülür.

```
UNIX> !ps
ps4 5 3 5 5 5 3
  0: user  0/000: Sleeping for  4 seconds
  0: user  1/000: Sleeping for 10 seconds
  0: user  2/000: Sleeping for  5 seconds
  0: user  3/000: Sleeping for  2 seconds
  0: user  4/000: Sleeping for  7 seconds
  0: prnt  0/000: ready to print
  0: prnt  0 blocking because the queue is empty
  0: prnt  1/000: ready to print
  0: prnt  1 blocking because the queue is empty
  0: prnt  2/000: ready to print
  0: prnt  2 blocking because the queue is empty
  2: user  3/000: Submitting a job with size 5
  2: user  3/001: Sleeping for 10 seconds
  2: prnt  0/000: Printing job   0 from user  3 size   5
  4: user  0/000: Submitting a job with size 1
  4: user  0/001: Sleeping for  1 seconds
```

```
 4: prnt  1/000: Printing job   0 from user  0 size   1
 5: user  2/000: Submitting a job with size 4
 5: user  2/001: Sleeping for  6 seconds
 5: user  0/001: Submitting a job with size 3
 5: user  0/002: Sleeping for 10 seconds
 5: prnt  2/000: Printing job   0 from user  2 size   4
 7: user  4/000: Submitting a job with size 4
 7: user  4/001: Sleeping for 10 seconds
 8: prnt  1/001: ready to print
 8: prnt  1/001: Printing job   1 from user  0 size   3
10: user  1/000: Submitting a job with size 1
10: user  1/001: Sleeping for  6 seconds
11: user  2/001: Submitting a job with size 3
11: user  2/002: Sleeping for  1 seconds
12: user  3/001: Submitting a job with size 1
12: user  3/002: Sleeping for 10 seconds
12: user  2/002: Submitting a job with size 5
12: user  2/003: Done
...
```

Bir hata

Ancak bu kodlarla ilgili halen bir problem vardır. Kuyruk boş olduğu için beklemede iki yazıcı iş parçacığı olduğunu varsayalım. Dahası aynı zamanda iş göndermek isteyen kullanıcı iş parçacığı da vardır. İlk kullanıcı iş parçacığı işi kuyruğa koyar ve **pthread_cond_signal()**'ı çağırır. Bu bloke olmayan yazıcı iş parçacıklarından biri ancak mutex'i yakalamak için bloke olabilir. Şimdi kullanıcı iş parçacığı mutex'i serbest bırakır, fakat yazıcı iş parçacığı alınmaz ve bunun yerine bir sonraki yeni kullanıcı iş parçacığını alır. Bu parçacık kuyruğa bir iş koyar fakat kuyrukta öncesinden koyulmuş bir iş olduğundan **pthread_cond_signal()** komutu çağrılamaz. Bu yüzden yazılmak için iki iş olmasına rağmen yalnızca bir yazıcı iş parçacığı canlıdır. Bu şu anlama gelir bir yazıcı kaybedildi ve bu da bir hatadır.

ps4-bad.txt' de durum tam olarak görülmektedir. Üç kullanıcı iş parçacığı ve beş yazıcı iş parçacığı vardır. Başlangıçta yazıcı iş parçacıklarının tamamı blokedir. 3. saniyede iki kullanıcı iş parçacığı iş gönderir fakat sadece bir yazıcı iş parçacığı (0 nolu iş parçacığı) işaret edilir. Daha sonra yazıcı kuyruğuna daha fazla iş koyulur fakat njobs 1'den büyük olduğundan artık yazıcılar canlandırılamaz. Bu durum da bir hatadır.
ps5.c' de de görüldüğü gibi **pthread_cond_signal()** da yer alan if ifadesini silmek, bu hatanın basit bir şekilde düzeltilmesini sağlar. Bu şu anlama gelmektedir **submit_job** daima boş durum değişkenlerini, **get_print_job** ise daima dolu durum değişkenlerini işaret etmektedir.
Bu yapı iyi bir şekilde çalışır—eğer bloke edilmiş iş parçacığı yoksa **pthread_cond_signal()** herhangi bir şey yapmaz ve eğer kullanıcı iş parçacığı bloke edilmemiş ise ve kuyrukta boş yer yoksa **pthread_cond_wait()** ifadesi tekrar çağırılacaktır. Bunu bir deneyin. Eğer **ps5-good.txt**'ye bakarsanız **ps4-bad.txt'** in 27. Dakikası ile aynı senaryo olduğunu görürsünüz

ve bu durum daha iyileştirilmiş şekildedir.

Sonuç

Böylece, monitör ve koşul değişkenlerinin ne olduğunu ve onların kullanımıyla ilgili detaylı örnekler görmüş oldunuz. Ve ayrıca senkronizasyon problemlerini ince bir zekayla planlamalı ve programlarınızın çıktılarının sizin istediğiniz gibi olduğundan emin olmak için çıktıları dikkatli bir şekilde gözlemlemelisiniz.