FORK

Unix'te tüm prosesler fork() sistem çağrısı ile oluşturulur. Bu istisnasız bir durumdur. Fork'un ne yaptığını aşağıda inceleyelim :

Çağırılan prosesin bir kopyası olarak yeni bir proses oluşturur. Bu şu anlama geliyor ; fork() , çağrılan prosesin belleğini (kod , globaller , heap ve stack) , registerlarını ve açık dosyalarını kopyalar. Çağrılan proses yeni oluşturulan prosesin pid'si ile fork()'tan geri döner ("child proses " çağrılır sonra çağrılan proses" parent prosesi " çağırır.). Yeni oluşturulan proses parent'ın bir çifti (kopyası) olarak fork() çağrısını geri döndürür. Bu geri dönen değer(return) sıfır değeridir. Bu değer fork()'un geri dönüş değerinde hangi prosesin olduğunu anladığımız değerdir.

Unix programlamada , fork oldukça masraflıdır. Bu yüzden thread kullanımı bölüm motivasyonunu arttırmak üzere hizmete girmiştir. Fakat prosedür olarak yeni bir proses oluştrma yıllar içinde ivme kazandırır, özellikle Linux dünyasında. Bir prosese fork işlemi yapma artık çok özel dikkat gerektirecek işlemlere maruz kalmayı kabul edemez(bazı nadir durumlar hariç). Birçok teknik bunu, başarmak için kullanmıştır, önemli COW (copy-on-write) olma metodu ile. COW'da ebeveynin(parent) verisi ,yığını ve yığıtı, parent ve child tarafından paylaşılır, ve sadece okunur çekirdek(kernel) tarafından değiştirilebilir koruma düzeyine sahiptir. Her iki proses bu bölgeleri değiştirmeye kalkarsa çekirdek, bellek biriminden bir kopya alır , özellikle bir sayfa. COW büyük bir kazançtır. Özellikle fork'u takip eden child prosesteki çağrıdaki etkisini bilmek büyük olasılıkla exec ile eşdeğerdir. B ir çağrı, bir diğer proses ile şimdiki prosesin üstüne yazmaktır. Diğer teknikler tarafından desteklen görüş , Linux proses spawn(yumurta) proses diğer işletim sistemlerinde thread spawndan daha etkili olabileceğini göstermiştir.

simpfork.c örneğimize bakalım :

```
main()
{
  int i;

  printf("simpfork: pid = %d\n", getpid());
  i = fork();
  printf("Did a fork.  It returned %d.  getpid = %d.  getppid = %d\n",
    i, getpid(), getppid());
}
```

Programı çalıştırdığımızda çıktı şu şekilde olur:

```
UNIX> simpfork
simpfork: pid = 914
Did a fork.  It returned 915.  getpid = 914.  getppid = 381
Did a fork.  It returned 0.  getpid = 915.  getppid = 914
UNIX>
```

Peki ne oluyor programda. simpfork çalıştırıldığı zaman pid 914 'tür. Daha sonra

fork() çağrısı pid si 915 ile bir kopya proses oluşturuyor. Ebeveyn(parent) CPU'nun kontrolünü kazanır ve fork()'tan 915 değerini geri döndürür. Bu yavru(child) prosesin pid'sidir. Yavru proses bu geri dönüş değerini yazdırır, onun kendi pid'si ve csh'ın pid'si , hala 381 . Sonra programdan çıkar. Daha sonra child CPU'yu alır ve fork()'tan 0 değeri ile geri döndürür. Değeri yazdırır, bu onun pid'si , ve parent'ın pid'si.

Not: fork() sistem çağrısından sonra CPU'nun kontrolünü hangi prosesin kazanacağının garantisi yoktur. Bu ebeveynde olabilir , yavruda. .simpfork'u ikinci kez çalıştırdığımızda yavru ilk kontrolü alır.

```
UNIX> simpfork
simpfork: pid = 928
Did a fork.  It returned 0.  getpid = 929.  getppid = 928
Did a fork.  It returned 929.  getpid = 928.  getppid = 381
UNIX>
```

Şimdide simpfork2.c'ye bakalım. O fork()'u çağırır ve parent hemen çıkış yapar.Yavru (child) proses sleep(5)'i çağırır. sleep(5) ; 5 saniye süreyle programımızı uykuya alır ve sonra child pid'si ekrana yazdırılır ve onun ebeveyninin (parent) pid'side.Zamanla child uykudan uyanır, parent programdan çıkar. Bu nedenle getppid(), parent'ın pid'sinin eski değerini geri döndüremez, pid artık geçerlidir. Unix bu gibi durumlarda yavru programın ebeveynliğine transfer yapıyor. Özellikle program çıkışında parent, init program (pid 1) onun ebeveyni olur. Böylece yavru (child) , ebeveynini ekrana yazdırdıktan sonra uykuya geçiyor. pid 1 çıktısı şu şekilde :

```
UNIX> simpfork2
Child.  getpid() = 1301, getppid() = 1300
Parent exiting now
UNIX> After sleeping.  getpid() = 1301, getppid() = 1
```

Şimdi, simpfork2.c ' e bakalım. Bu fork() fonksiyonunu çağırır ve ana çıkışa sahiptir. Child sleep(5) fonksiyonunu çağırır, bu beş saniye boyunca uyutur ve sonra onun pid' sini, ve ebeveynin pid'sini yazdırır. Bunun hakkında püfnokta ne? Peki, zamanla çocuk uykudan uyanır, evebeyn çıkıldı ve onun süreci biter. Böylece, getppid() evebeynin pid'sinin eski değerini geri döndürmez, bu pid artık geçerli değildir. Hangi Unix bu durumlarda çocuğun programının evebeynliğini transfer eder. Özellikle, program çıktığında, init(pid 1) programı onun evebeyni olur. Böylece, evebeyni uyuduktan sonra child yazdığında, o pid 1 yazdırır:

```
UNIX> simpfork2
Child.  getpid() = 1301, getppid() = 1300
Parent exiting now
UNIX> After sleeping.  getpid() = 1301, getppid() = 1
```

Unutmayın “UNIX>” child hala çalışıyor olsa bile istem döner bir kez evebeyn döner.Bunun nedeni csh sadece evebeynin bitirmesini bekler,diğer süreçler için beklemez.

Simpfork3.c fork boyunca evebeynin adres alanını child’a kopyalayan basit bir programdır. Forktan sonra her proses j ve K için bellek bölgesine sahiptir. Böylece, child’ ın j değeri 201, K değeri 301’ e değiştiğinde; sadece child’ın j ve K değeri değişir, evebeynin değeri değişmez:

```
UNIX> simpfork3
Before forking: j = 200, K = 300
After forking, child: j = 201, K = 301
After forking, parent: j = 200, K = 300
UNIX>
```

İlginçtir, eğer biz simpfork3’ün çıkışını bir dosyaya yönlendirirsek, aşağıdaki koda bakalım:

```
UNIX> simpfork3 > output
UNIX> cat output
Before forking: j = 200, K = 300
After forking, child: j = 201, K = 301
Before forking: j = 200, K = 300
After forking, parent: j = 200, K = 300
UNIX>
```

Bu kitapta anlatılanları ben burada açıklayacağım. Yönlendirme terminale çıktı ürettiğinde, stdout satır satır tamponlanır – bir kere putchar(‘\n’) veya dengini yap, geçici belleğe standart çıkış ile write(1,…) yazdırılır. Ancak stdout bir dosyaya yönlendirildiğinde, stdio kütüphanesi tamponlar – bazı geniş tamponlar(muhtemelen 4k veya 8K karakterler) dolana kadar yazmaz. Böylece fork çağrıları zamanında, ”Before forking:” yazısı fd=1’ e yazılmaz. Yerine bu standart giriş/çıkış kütüphanesinde tamponlanır. Fork çağrıldığında, bu geçici bellek simpfork3’ün adres alanın bir kısmıdır böylece child prosese kopyalanır. Böylece baytlar geçici bellekten temizlendiğinde dosyaya iki kere “Before forking:” yazılır. Bu anlamak için önemli bir konudur. Tuhaf görünüyor ama mantıklı bir açıklaması var.

Simpfork4.c fork çağrılarının açık dosyaları nasıl paylaştığını gösterir.Özellikle tüm açık dosya tanımlayıcıları çoğaltılamaz. Böylece yazılmak için simpfork4.c de açılan dosya hem evebeyn hem de child proseste açılır.Açık dosyanın aynı olduğunu unutmayın – bir dosyanın sonuna yazarken, seek işaretçisi evebeyn ve child’da değişir. Bunun nedeni işletim sisteminde her iki prosesinde aynı açık dosyayı işaret etmesidir. Eğer iki proseste aynı açık dosya işaretçisini paylaşmıyorsa, evebeyn Child’ın yazdığının üzerine yazabilir:

```
UNIX> simpfork4
```

```
UNIX> cat tmpfile
Before forking
Child: After forking: Seek pointer = 15
Parent: After forking: Seek pointer = 55
UNIX>
```

Bu ayrıca kitapta açıklanmıştır.

Dup derslerinde dosya tanımlayıcıları paylaşımı hakkında daha fazla tekrar edeceğim.