Bir Unix Programcısı nasıl olur

Farklı kişilerin "Unix programcı" terimiyle ilgili farklı anlayışları vardır. Birçok insan için sadece Unix işletim sistemi üzerinde program yapabilen bir programcı anlamına gelir. Bu, gerçekten bu anlama gelmez,çoğu durumda taşınabilir kodları yazmak istiyoruz çünkü bu standart C veya C++ ve küçük Unix özel istem çağrıları,vb ile ilgisi var.

Profesyoneller için bir Unix programcı olmak oldukça farklı bir anlam ifade eder ve programlamanın da ötesine geçer. "Unix mühendisi" terimini kullanmak bu programlamayı yaparken bir mühendislik yapılması gerektiğini vurgulamaktır. Ama bu terim biraz komik, o yüzden Unix programcı özgün ifade ile kalmasına izin verir. Hatırlamanız gereken sadece bu önemli ayrım. Bu nedenle, sistem çağrıları hakkında her şeyi bilen teknik bir Unix programcı olmaz.

Görünümünde bir mühendislik açısından bakıldığında, gerçek Unix programcılar çoğunlukla temel araçları bir API yoluyla diğer insanlar tarafından yeniden kullanılmasını sağlayan, yazılım geliştirmeyi amaçlayan bir topluluk. Unix programcı olmak tüm yönlerini kapsayacak şekilde, Eric Raymond tarafından yazılmış "Unix Programlama Sanatı"kitabını, okumalısınız. Burada ise URL kitabın online sürümü. Burada, bizim ders tamamlarken, Raymond'ın kitabında birkaç önemli nokta üzerine gitmek ve keşfetmek için yeni kapılar açmaya çalışacağız.

Unix programcılarının dört önemli özellikleri vardır. Birincisi, çok yüksek seviyeli dilleri kullanırlar. Onlar çeviriciyi(assembly) değiştirmek için C dilini icat edenlerdir. Ayrıca C değiştirmek için geliştirilen Perl gibi olanlardır. İkincisi, çoğunlukla veri güdümlü programlama metodolojisi uygulamak ve büyük ölçüde nesne yönelimli programlamadan nefret ederler. Veritabanlı program yaparken,biri açıkça veri yapılarını kodlardan ayıran özellikleri hatta tasarımları da görür,böylece biri sadece kodla değil veri yapısını düzenleyerek programın mantığında değişiklikler yapabilir. Verilere bağlı (DD) ve nesne yönelimli (OO) bu birbirinden farklıdır:

a. bir DD, verilerin salt bir nesne olmaması durumudur

ama aslında programın akış kontrolünü tanımlar

b. OO temel kaygısı, birincil endişesi kapsülleme olduğundan

DD mümkün olduğunca az sabit kod olarak yazıyor

Gibi çoğu teyit durumunda, is-a ve has-a ilişkileri tanıtıldı OO ile yüksek miktarda karmaşıklık geldi bu normal,biz görmek istemiyoruz ve iyi tanımlanmış bir API verilirse gerçekten görmeyiz. Karmaşık sistemler oluştururken güvenilir, genişletilebilir, kullanımı kolay olması çok önemlidir. İşletim sistemleri ve geliştirme ortamları bu kategoride düşmesine neden olur.

Üçüncüsü, kod geliştiricileri kullanmaları mümkün. İyi örnekler aslında gerçek bir kod satır yazarak lex ve yacc kullanılarak geliştirilebilir. Dördüncü, Mini dillerin özel etki alanı.Yazılımdaki hata örüntülerinin büyük ölçekli çalışmalarından elde edilen tutarlı sonuçlar biri, programcının yüzlerce satır başına hata oranlarının azalması, programcılar ve kodlama dilinin bağımsız olmasıdır.Yüksek seviyeli dillerin,az satırda yazılması için izin vermesi, daha az hata demektir. Unix, büyük ölçüde programların satır sayısını azaltmak için uzmanlaşmış küçük dilleri barındıran uygulama alanına sahiptir. Domain- spesifik dil örnekleri (yapmak, yacc, lex) çok sayıda Unix dizgi dilleri (troff, eqn, tbl, pic, grap), kabuk programları (awk, sed, dc, bc) ve yazılım geliştirme araçları içerir.

Domain -spesifik küçük dillerin son derece güçlü bir tasarım fikri vardır. Görev için programcılar ,uygun yöntemler, kurallar ve algoritmalar belirtmek için kendi leri üst düzey dil tanımlayarak karmaşıklığı üst seviyeye iterler. Bu, aynı amaçlar için kablolu daha düşük seviyeli kodu kullanan bir tasarıma göre küresel bir karmaşıklık önemli bir azalmaya neden olur. İyi bir minilanguage tasarımı konusu ve en önemlisi bir minilanguage cs360 ötesinde ihtiyacınız olan bir minilanguage nasıl tasarlanır .Yeterli anlamda aydınlanmak için Raymond kitabından yararlanmalısınız. Kısacası, her uygulama alanında etki alanı, ilkel ,basit ve kalıplaşmış olanlar ,kullanıcıların muhtemel olarak tercih ettiğidir. Hangi yollarla akışkan ve değişken olup, bir minilanguage yararlı olacaktır. Yukarıdaki örneklerin yanı sıra, bu postscript fark da yazıcılar için üretmek her *. Ps dosya gerçekten tabii, çok sayıda veri içeren bir minilanguage programdır.

Durum Kontrolleri

Çok sık duyduğumuz "hey, neden kodunuzu bazı sağlık kontrolleri çalıştırmak değil mi?".Sonra şaşkın bir yüz görürsünüz. Bu profesyonel bir yazılım mühendisi olmak bir nebze(ciddi) ile ilgilidir. Genel olarak iyi bir gelişme uygulama şunlar olmalıdır:

- Bazı olmadıkça özel kodları güvenmeyin
  ezici bir çoğunlukla iyi neden
- Taşınabilir olmak için kod yazmak, örneğin GNU autotools kullanın

- Yayınlanmadan önce iyice kodu test (bu bile bir soru olmalıdır?)
- Kodunuzu aklı-kontrol hale getirin

"gcc-Wall"

- Bellek sızıntıları bakmak ve araçları çalıştırmak
  diğer çalışma zamanı hataları (elektrikli çit, Valgrind, checkergcc, mcheck, MPR,vb)
- Python projeleri için, PyChecker çalıştırın
  sourceforge.net / projects / pychecker
- Perl, (belki-T, varsa) perl-c ile kodunuzu kontrol
  perl-w kullanmak ve dinsel "katı kullanın"

Size bir örnek sağlamlık denetimi araçları bir tat vermek için, en Eclectic Çit kısaca bir göz atalım. Herhangi bir linux kutusu üzerinde, adam libefence yazın ve Elektrik Çit çok iyi yazılmış bir açıklama görebilirsiniz. Bu hafıza problemleri şu tür, globals, yığın veya yığınolursa olsun bakmak için kullanılır:

bellek sızıntısı: free'ed değil malloc'ed bellek
bellek taşmaları: Bir malloc'ed tampon sonunda belleğe erişim
bellek underruns: Bir malloc'ed tampon başlamadan önce belleğe erişim
Diğer: örn Zaten free'ed bir tampon erişen

Elektrik Çit normal malloc ve özel sürümleri ücretsiz çağrılar değiştirerek çalışır. Elektrik Çit hemen sonra ulaşılmaz bir bellek sayfası yerleştirmek için bilgisayarınızın sanal bellekdonanım kullanır (veya daha önce, kullanıcının tercihine bağlı olarak) her bir bellek allo-katyon. Yazılım okur veya kusurlu talimatıyla program durdurma, bu erişilmez sayfa,donanım sorunları bir segment hataya yazar zaman. En sevdiğiniz debugger kullanarakhatalı deyimi bulmak için daha sonra triv-ial olduğunu. (Ücretsiz serbest bırakıldı benzer bir şekilde, bellek) erişilmez yapılır ve dokunduğu herhangi bir kod bir segment hatayaalacak. Elektrik Çit kullanarak Major sorunların biri kaynak tüketimi. Bildiğimiz gibi, enmoderm donanım ayırır ve bir sayfa (ki, 4KB veya 8KB, vb) temelinde bellek korur.Elektrik Çit temelde her bir tahsisat bir sayfa üzerinde ikamet ve taşması ve boşalması, vb böylece bellek hatası neden ayrı bir sayfada devam edecek gibi tampon bulmak olun.Elektrik Çit kullanarak sadece yapmak çok basittir:

```
gcc-o myexecutable .... -lefence
```

Eğer varsa Sonra myexecutable çalışan tüm bellek hataları çıktı olurdu. Gdb ilekullanarak, bellek hata hakkında ayrıntılı bilgi bulunabilir.
Örneğin, broken.c düzgün çalışmalıdır bir programdır. Bunu tam olarak neden herkesinbilmek bekliyoruz. Biz sağlamlık denetimi çalıştırmak nasıl bir göz atın sonra atalım:

```
UNIX> cat broken.c
#include
#include

main()
{

char * s;
s = malloc(5);
strcpy(s,"what hu");
printf("%s\n", s);
free(s);
}
UNIX> gcc broken.c
UNIX> a.out
what hu
UNIX> gcc broken.c -lefence
UNIX> a.out

Electric Fence 2.2.0 Copyright (C) 1987-1999 Bruce Perens
Segmentation fault(Parçalama hatası)
UNIX> gcc -g broken.c -lefence
UNIX> gdb a.out
GNU gdb Red Hat Linux (5.2-2)
Copyright 2002 Free Software Foundation, Inc.

GDB, GNU (Genel Kamu Lisansı) tarafından kapalı ücretsiz bir yazılımdır ve değiştirmek
ve\/veya belirli koşullar altında kopyaları dağıtmayı bekliyoruz.

Koşulları görmek için "kopyalama göster" yazın.

GDB  için kesinlikle hiçbir garantisi yoktur. Detaylar için "Garanti ayrıntılarını gösterme"
yazın.
Bu gdb olarak yapılandırılmış "i386-redhat-linux " gibi...

(gdb) r

Başlangıç programı: /home/huangj/a.out
[Yeni konu 1024 (LWP 30991)]
```

Electric Fence 2.2.0 Copyright (C) 1987-1999 Bruce Perens

Program sinyali SIGSEGV, Segmentation fault(Parçalama hatası) aldı.

[Thread(konuya) ‘ e aktarma 1024 (LWP 30991)]

0x420807b6 strcpy ()‘ nin içinde /lib/i686/libc.so.6 ‘dan
(gdb) Burada

#0 0x420807b6 strcpy ()‘ nin içinde /lib/i686/libc.so.6 ‘dan
#1 0x080485f8 main ()‘ nin içinde broken1.c:9 ‘ da
#2 0x42017589 __libc_start_main ()’nin içinde /lib/i686/libc.so.6 ‘dan

---

Open Source(Açık Kaynak)

Unix Programcılar Open Source(Açık Kaynak) hakkında çok şey bilmektedir. Bildiğimiz gibi, Unix efsanesi 70 lerde açık kaynak(open source) çabası olarak başaladı, ve başarılı bir şekilde devam etti, ta ki, açık kaynak(open source) tökezleyene ve tutulamaz hale gelene dek, ozaman mal sahipleri de paylarına düşeni yaptı. Tabii ki, “Açık Kaynak(Open Source)” dönemi son yüzyılın sonuna kadar gelişmiş değildir.

Modern bir UNIX programcısı olan biri, en azından açık kaynak hakkında bazı temel noktalarını, ve çoğunlukla da bir yazılım lisansının yayınlamasından (piyasaya sürülmesinden) anlamalıdır. Lisans süresi geldiğinde; kopyalama ve yeniden dağıtma hakları, kullanma hakları, kişisel kullanımda değişiklik yapmak için haklar, ve kopyalardaki değişiklikleri yeniden dağıtmak için haklar vardır.

Açık Kaynak tanımı (www.opensource.org/osd.html) lisanslama üzerindeki kısıtlamaları aşağıdaki şartları empoze eder:

- Kopyalamak için sınırsız haklar verilebilir.

- Değiştirilmemiş haliyle yeniden dağıtmak için sınırsız bir hakkı verilebilir.

- Kişisel kullanım için değiştirmek için sınırsız bir hakkı verilebilir.

Kurallar modifiyeyi ve ikili yeniden dağıtılmassında kısıtlamaları yasaklar, bu bağlantısız çalışma kodları sevk edebilmek için gereken yazılım distribütörlerin ihtiyaçlarınızı karşılar. Bu yazarlar değiştirilmiş kaynakları böylece yazarın niyet ve başkaları

tarafından herhangi bir değişiklik bir "denetim izi" kurulması, bakir kaynakları artı yamalar olarak dağıtılması olarak istemesine olanak sağlar.

Standart lisansların tümü (MIT ve ya X Consortium License, BSD, Artistic, GPL/LGPL ve Mozilla Public License) tıR.