2. Niti

Niti, kao i procesi, su mehanizam koji omogućava da program radi više od jedne stvari istovremeno. Kao i kod procesa, niti se izvršavaju istovremeno; Linux kernel ih raspoređuje asinhrono, prekidajući svaku nit sa vremena na vreme, da bi ostale imale priliku da se izvrše.

Konceptualno, nit postoji unutar procesa. Niti su finije izvršne jedinice od procesa. Kada pozivate program, Lunux kreira novi proces i u tom procesu se stvara jedna nit koja izvršava program sekvencijalno. Ta nit može da stvori dodatne niti; sve ove niti izvršavaju isti program u istom procesu, ali svaka nit može da izvršava različit deo programa u ma kom vremenu.

Videli smo kako program može da izvrši funkciju fork nad dete procesom. Dete proces inicijalno izvršava svoj roditeljski proces, sa njegovom roditeljskom virtualnom memorijom, deskriptorima datoteka, i tako dalje. Dete proces može da modifikuje memoriju, zatvori deskriptore datoteka , i tome slično, bez uticaja na svoj proces, a važi i obrnuto. Ipak, kada program kreira novu nit ništa nije kopirano. Stvarajuća i stvorena nit dele isti memorijski prostor, deskriptore datoteka, i druge sistemske resurse kao i original. Ako jedna nit promeni vrednost promenjive, druga nit će potom videti izmenjenu vrednost. Slično tome, ako jedna nit zatvori deskriptor, druge niti neće moći da pročitaju ili da upišu u taj deskriptor datoteke. Zbog toga što proces i sve njegove niti mogu da izvršavaju samo jedan program istovremeno, ako ma koja nit unutar procesa pozove jednu od exec funkcija, sve ostale niti se ukidaju (novi program može, na primer, da stvori nove niti). GNU/Linux ostvaruje POSIX standard niti API (poznat kao PThread). Sve funkcije niti i tipovi podataka su deklarisane u header datoteci <pthread.h>. Pthread funkcije nisu uključene u standardnu C biblioteku. Umesto toga, one se nalaze u libthread, tako da treba da dodate –lpthread u komandnoj liniji kada pozivate vaš program.

2.1. Kreiranje niti

Svaka nit u procesu se identifikuje pomoću identifikatora niti (thread ID). Kada je u pitanju Identifikator niti u C ili C++ programima, koristite pthread_t tip.

Čim se završi kreiranje, svaka nit izvršava nitsku funkciju. Ovo je obična funkcija i sadrži kod koji nit treba da izvrši. Na GNU/Linux sistemu, nitske funkcije uzimaju jedan parametar, void* tipa, i imaju vraćenu vrednost tipa void*. Parametar je argument niti: GNU/Linux prosleđuje vrednost do niti, bez gledanja u njega. Vaš program može da koristi ovaj parametar da prosledi podatke novoj niti. Slično tome, vaš program može da koristi vraćenu vrednost da bi prosledio podatke iz niti koja se završava ka njenom tvorcu.

Niti (Threads)

Funkcija pthread_create stvara novu nit.

Možete je snabdeti sledećim:

1. Pokazivač na pthread_t promenjivu, u kojem je Identifikator niti za novu nit smešten.
2. Pokazivač na atribut niti, objekat. Ovaj objekat kontroliše detalje o tome kako nit vrši interakciju sa ostatkom programa. Ako prosledite vrednost NULL kao atribut niti, nit će biti kreirana sa uobičajenim atributm niti. Atributi niti su opisani u poglavlju 4.1.5. Atributi niti.
3. Pokazivač na funkciju niti. Ovo je običan pokazivač na funkciju, tipa: void* (*) (void*)
4. Vrednost argumenta niti tipa void*. Šta god da prosleđulete, ono se jednostavno prosleđuje kao argument funkciji niti kada nit počne da se izvršava.

Poziv pthread_create momentalno vraća, a originalna nit nastavlja sa izvršavanjem instrukcija sledećeg poziva. U mađuvremenu, nova nit počinje sa izvršavanjem funkcije niti. Linux vrši raspoređivanje obe niti asinhrono, i vaš se program ne sme oslanjati na relativan raspored po kome se instrukcije izvršavaju u okviru dve niti.

Program u primeru 2.1. stvara nit koja pokazuje X u kontinuitetu do standardne greške. Nakon pozivanja thread_create, glavna nit prokazuje O u kontinuitetu do standardne greške.

Primer 2.1. (thread-create.c) Stvaranje Niti

```
#include <pthread.h>
#include <stdio.h>

/* Ispisuje X do stderr. Parametar nije koriscen. Ne vraca. */

void* print_xs (void* unused)
{
      while (1)
      fputc ('x', stderr);
      return NULL;
}

/* Glavni program */

int main ()
{
      pthread_t thread_id;
      /* Stvara novu nit. Nova nit izvrsava print_xs funkciju */

      pthread_create (&thread_id, NULL, &print_xs, NULL);

      /* Ispisuje O u kontinuitetu sve do stderr. */
      while (1)
      fputc ('o', stderr);
```

```
      return 0;
}
```

Sastavite i proverite ovaj program koristeći sledeći kod:

```
% cc –o thread-create thread-create.c –lpthread
```

Pokušajte da izvršite program da biste videli šta se dešava. Primetite da je šema X i O nepredvidiva, kada Linux alternativno raspoređuje dve niti.

Pod normalnim okolnostima, nit izlazi na jedan od dva načina. Jedan način, kako je ranije pokazano, jeste vraćanjem iz funkcije niti. Vraćena vrednost iz funkcije niti je izabrana da bude vraćena vrednost iz niti. Naizmenično, nit može da izađe eksplicitno pozivanjem pthread_exit. Ova funkcija može biti pozvana od strane funkcije niti ili neke druge funkcije pozvane direktno ili indirektno od strane funkcije niti. Argument za funkciju pthread_exit je vraćena vrednost niti.

2.1.1. Prosleđivanje podataka niti

Argument niti obezbeđuje pogodan metod za prosleđivanje podataka niti. Zbog toga što je tip argumenta tipa void, ne može se proslediti puno podataka direktno preko argumenta. Umesto toga koristite argument niti da biste prosledili pokazivač nekoj strukturi ili nizu podataka. Jedna, uobičajena tehnika je da se definiše struktura za svaku funkciju niti, koja sadrži "parametre" koje funkcija niti očekuje.

Korišćenjem argumenta niti, na lak način se može ponovo upotrebiti ista funkcija niti za mnoge niti. Sve ove niti izvršavaju isti kod, ali nad različitim podacima. Program u primeru 2.2. je sličan prethodnom primeru. Ovaj stvara dve nove niti, jednu koja ispisuje X a drugu koja ispisuje O. Umesto toga da se ispisuje beskonačno, svaka nit ispisuje utvrđen broj karaktera i potom izlazi, vraćajući se iz funkcije niti. Istu funkciju niti, char_print, koriste obe niti, ali svaka je konfigurisana drugačije, korišćenjem strukture char_print_parms.

Primer 2.2. (thread-create2) Stvaranje dve niti

```
#include <pthread.h>
#include <stdio.h>

/* Parametri ka print_function */

struct char_print_parms
{
       /* Karakter koji se ispisuje */
```

```
      char character;
      /* Broj puta koliko se ispisuje */
      int count;
};

/* Ispisuje broj karaktera za stderr, koji su PARAMETRI, koji je pokazivac na
strukturu char_print_parms. */

void* char_print (void* parametres)
{

      /* Pravi cookie pokazivac na odgovarajuci tip */

      struct char_print_parms* p = (struct char_print_parms*) parametres;
      int i;

      for (i = 0; i < p -> count; ++i)
      fputc (p->character, stderr);
      return NULL;
}

/* Glavni program */

int main ()
{
      pthread_t thread1_id;
      pthread_t thread2_id;
      struct char_print_parms thread1_args;
      struct char_print_parms thread2_args;


      /* Stvara novu nit koja ispisuje 30.000 X */
      thread1_args.character = 'x';
      thread1_args.count = 30000;
      pthread_create (&thread1_id, NULL, &char_print, thread1_args.);

      /* Stvara novu nit koja ispisuje 20.000 O */
      thread2_args.character = 'o';
      thread2_args.count = 20000;
      pthread_create (&thread2_id, NULL, &char_print, thread2_args.);

      return 0;
}
```

Ali stanite! Program u primeru 2.2. ima ozbiljnu grešku u sebi. Glavna nit (koja izvršava funkciju main) stvara strukture parametara niti (thread1_args i thread2_args) kao lokalne promenjive, i onda prosleđuje pokazivače na ove strukture nitima koje stvara. Šta sprečava Linux da raspoređuje tri niti na takav način da main završi izvršavanje pre nego što se bilo koja od druge dve niti završe? Ništa! Ali ako se to dogodi, memorija koja sadrži strukture parametra niti će biti vraćena, dok joj druge dve niti još uvek pristupaju.

2.1.2. Združivanje niti

Jedno rešenje je primorati main da čeka dok druge dve niti završe. Ono što nama treba jeste funkcija koja je slična funkciji wait, koja čeka da se nit završi umesto procesa. Ta funkcija je pthread_join, koja uzima dva argumenta: Identifikator niti za nit koja se čeka, i pokazivač na void* promenjivu koja će primiti krajnju vrednost niti koja se vraća. Ukoliko vas ne zanima vraćena vrednost niti, prosleđuje se NULL kao drugi argument. Primer 2.3. pokazuje ispravljenu main funkciju iz pogrešnog primera 2.2. U ovom slučaju, main ne postoji sve dok obe niti koje ispisuju X i O završe sa radom, tako da više ne koriste argumente strukture.

Primer 2.3. Ispravljena main funkcija za thread-create2

```
int main ()
{
      pthread_t thread1_id;
      pthread_t thread2_id;
      struct char_print_parms thread1_args;
      struct char_print_parms thread2_args;


      /* Stvara novu nit koja ispisuje 30.000 X */
      thread1_args.character = 'x';
      thread1_args.count = 30000;
      pthread_create (&thread1_id, NULL, &char_print, thread1_args.);

      /* Stvara novu nit koja ispisuje 20.000 O */
      thread2_args.character = 'o';
      thread2_args.count = 20000;
      pthread_create (&thread2_id, NULL, &char_print, thread2_args.);

      /* Provera da je prva nit zavrsena */
      pthread_join (thread1_id, NULL);
      /* Provera da je druga nit zavrsena */
      pthread_join (thread2_id, NULL);

      /*Sada je vracanje bezbedno*/
      return 0;
}
```

2.1.3. Povratne vrednosti niti

Ako je drugi argument koji prosleđujete funkciji pthread_join različit od null, vraćena vrednost niti će biti smeštena na lokaciji na koju ovaj argument pokazuje. Vraćena vrednost niti, kao i argument niti, je

tipa void*. Ako želite da povratite neki int ili neki drugi mali broj, možete to lako uraditi tako što ćete odrediti da vrednost bude void*, a onda ponovo odrediti da vrednost bude odgovarajućeg tipa nakon pozivanja funkcije pthread_join.[1]

Program u primeru 2.4. izračunava n-ti prosti broj u zasebnoj niti. Ta nit vraća željeni prosti broj kao svoju vraćenu vrednost niti. Glavna nit je, u međuvremenu, slobodna za izvršavanje drugog koda. Primetite da je uzastopna podela algoritma korišćena u compute_prime prilično neefikasna; konsultujte neku knjigu koja govori o numeričkim algoritmima ako treba da izračunavate puno prostih brojeva u svom programu.

Primer 2.4. (primes.c) Izračunavanje prostih brojeva u niti

```
#include <pthread.h>
#include <stdio.h>

/* Izracunavanje uzastopnih prostih brojeva (vrlo neefikasno).
   Vracanje Ntog prostog broja, gde je N vrednost ukazana u *ARG */

void* compute_prime (void* arg)
{
      int candidate  = 2;
      int n = *((int*) arg);

      while (1) {
            int factor;
            int is_prime = 1;

            /* Testiranje prostih brojeva pomoću deljenja */

            for (factor = 2; factor < candidate; ++factor)
                  if (candidate % factor ==0 ) {
                        is_prime = 0;
                        break;
                  }
                  /* Da li je ovo prost broj koji trazimo? */
            if (is_prime){
                  if(--n == 0)
                  /* Vrati zeljenji prosti broj kao vracenu vrednost niti */
                  return (void*) candidate;
            }
            ++candidate;
      }
      return NULL;
}

int main ()
{
      pthread_t thread;
      int which_prime=5000;
      int prime;
```

```
      /* Pocinje izracunavanje niti, sve do 5000tog prostog broja */
      pthread_create (&thread, NULL, &compute_prime, &which_prime);
      /*Obavljanje nekog posla ovde...*/
      /* Cekanje da se nit za prost broj zavrsi, a potom da se uzme rezultat */

      pthread_join (thread, (void*) &prime);
      /*Ispisuje se najveći prosti broj koji je izracunat */
      printf ("The %dth prime number is %d.\n", which_prime, prime);
      return 0;
}
```

2.1.4. Detaljnije o Identifikatorima niti

S vremena na vreme, korisno je da sekvenca koda odredi koja nit je izvršava. Funkcija pthread_self vraća identifikator niti one niti u kojoj je on pozvan. Ovaj identifikator niti se može upoređivati sa drugim identifikatorom niti pomoću funkcije pthread_equal.

Ove funkcije mogu biti korisne za određivanje da li određeni identifikator niti odgovara tekućoj niti. Na primer, pogrešno je da nit pozove funkciju pthread_join da bi sama sebe pridružila. (U ovom slučaju, funkcija pthread_join bi vratila grešku sa kodom EDEADLK.) Da biste ovo prethodno proverili, možete koristiti kod poput ovog:

```
if (!pthread_equal (pthread_self (), other_thread))
pthread_join (other_thread, NULL);
```

---

[1] Primetite da ovo nije prenosivo, i na vama je da osigurate da se vaša vrednost može bezbedno usmeriti na void* i natrag bez gubljenja bitova.

Vežbe:

Ukucajte, pokrenite i objasnite sledeće programe:

thread1.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>

void *thread_function(void *arg);

char message[] = "Hello World";
```

```
int main() {
    int res;
    pthread_t a_thread;
    void *thread_result;
    res = pthread_create(&a_thread, NULL, thread_function, (void *)message);
    printf("Waiting for thread to finish...\n");
    res = pthread_join(a_thread, &thread_result);
    printf("Thread joined, it returned %s\n", (char *)thread_result);
    printf("Message is now %s\n", message);
    exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
    printf("thread_function is running...\n”);
    sleep(3);
    strcpy(message, "Bye!");
    pthread_exit("Thank you for the CPU time");
}
```

thread2.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>

void *thread_function(void *arg);
int run_now = 1;
char message[] = "Hello World";

int main() {
    int res;
    pthread_t a_thread;
    void *thread_result;
    int print_count1 = 0;

    res = pthread_create(&a_thread, NULL, thread_function, (void *)message);

    while(print_count1++ < 20) {
        if (run_now == 1) {
            printf("1");
            run_now = 2;
        }
        else {
            sleep(1);
        }
    }

    printf("\nWaiting for thread to finish...\n");
    res = pthread_join(a_thread, &thread_result);
    printf("Thread joined\n");
    exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
    int print_count2 = 0;
```

```
    while(print_count2++ < 20) {
        if (run_now == 2) {
            printf("2");
            run_now = 1;
        }
        else {
            sleep(1);
        }
    }

    sleep(3);
}
```

thread3.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

void *thread_function(void *arg);
sem_t bin_sem;

#define WORK_SIZE 1024
char work_area[WORK_SIZE];
int main() {
    int res;
    pthread_t a_thread;
    void *thread_result;

    res = sem_init(&bin_sem, 0, 0);
    res = pthread_create(&a_thread, NULL, thread_function, NULL);
    if (res != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }
    printf("Input some text. Enter 'end' to finish\n");
    while(strncmp("end", work_area, 3) != 0) {
        fgets(work_area, WORK_SIZE, stdin);
        sem_post(&bin_sem);
    }
    printf("\nWaiting for thread to finish...\n");
    res = pthread_join(a_thread, &thread_result);
    printf("Thread joined\n");
    sem_destroy(&bin_sem);
    exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
    sem_wait(&bin_sem);
    while(strncmp("end", work_area, 3) != 0) {
        printf("You input %d characters\n", strlen(work_area) -1);
        sem_wait(&bin_sem);
    }
```

```
    pthread_exit(NULL);
}
```

thread3a.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

void *thread_function(void *arg);
sem_t bin_sem;

#define WORK_SIZE 1024
char work_area[WORK_SIZE];

int main() {
    int res;
    pthread_t a_thread;
    void *thread_result;

    res = sem_init(&bin_sem, 0, 0);
    res = pthread_create(&a_thread, NULL, thread_function, NULL);
    if (res != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }

    printf("Input some text. Enter 'end' to finish\n");
    while(strncmp("end", work_area, 3) != 0) {
      if (strncmp(work_area, "FAST", 4) == 0) {
        sem_post(&bin_sem);
        strcpy(work_area, "Wheeee...");
      } else {
        fgets(work_area, WORK_SIZE, stdin);
      }
      sem_post(&bin_sem);
    }

    printf("\nWaiting for thread to finish...\n");
    res = pthread_join(a_thread, &thread_result);
    printf("Thread joined\n");
    sem_destroy(&bin_sem);
    exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
    sem_wait(&bin_sem);
    while(strncmp("end", work_area, 3) != 0) {
        printf("You input %d characters\n", strlen(work_area) -1);
        sem_wait(&bin_sem);
    }
    pthread_exit(NULL);
}
```