MODUL PRAKTIKUM SISTEM OPERASI

PRAKTIKUM IV

Pembuatan Proses II

A. T U J U A N

Memahami mekanisme pembuatan proses child dan interaksinya dengan parent

B. DASAR TEORI

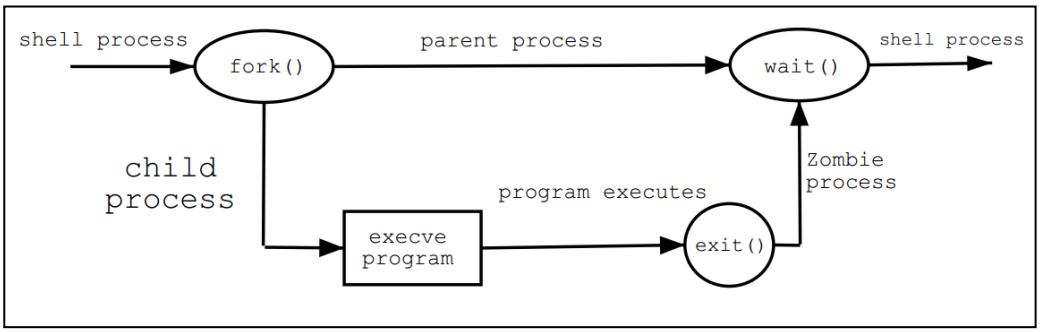
Model manajemen proses di Unix terbagi menjadi dua operasi:

1. Pembuatan proses
2. Jalannya program baru

Pembuatan proses baru dilakukan dengan perintah system call `fork()` dan program baru akan berjalan dengan system call keluarga `exec(l, lp, le, v, vp)`.

Ini merupakan fungsi-fungsi yang dapat digunakan secara terpisah, pemanggilan `fork()` akan membuat sub-process lengkap yang terpisah, yang akan sama persi dengan induknya. Sebaliknya pemanggilan keluarga `exec` akan menterminasi program yang berjalan saat ini dan memulai eksekusi proses baru yang benar-benar berbeda dengan induknya.

Alasan utama penggunaan model ini adalah untuk menyederhanakan operasi, ketika membuat sub-proses baru, lingkungan saat ini dari induk digunakan sehingga programmer tidak perlu mengatur lingkungan baru untuk menjalankan program. Setelah panggilan `fork`, program kemudian dapat menggunakan system call untuk melakukan modifikasi lingkungan yang sesuai dengan proses anak dan kemudian menggunakan fungsi `exec` untuk menjalankan proses baru yang dibutuhkan.



Gambar 1. Cara kerja perintah fork

Proses baru yang dibuat menggunakan fork disebut proses anak (child process). Fungsi ini dipanggil sekali namun memberikan dua return, yaitu pada proses anak mendapat return dengan nilai 0, dan pada proses induk mendapat nilai return sesuai ID proses anak yang baru. Alasan proses ID anak diberikan ke induknya karena suatu proses dapat memiliki lebih dari satu anak,

sehingga tidak ada fungsi tersedia untuk memperoleh proses ID anaknya.

Alasan fork mengembalikan nilai 0 ke proses anak karena suatu proses hanya boleh mempunyai induk tunggal, sehingga anak dapat memanggil `getppid` untuk memperoleh ID induknya.

Baik induk maupun anak melanjutkan eksekusi dengan instruksi yang mengikuti panggilan `fork`. Anak adalah salinan dari induknya. Sebagai contoh, anak memperoleh salinan dari *data space*, *heap*, dan *stack* milik induknya. Ini artinya ketika pertama kali anak dibuat, dia sama persis dengan induknya.

Jika induk dimatikan sebelum anak memanggil `exit()` untuk keluar secara normal maka akan menghasilkan proses zombie. Proses ini diturunkan oleh proses `init` yang merupakan bagian dari boot pada sistem Unix. Proses-proses ini harus dimusnahkan oleh user root yang memiliki fungsi init.

C. REFERENSI

Jonathan Macey. **Linux Systems Programming**. 2005.

D. LANGKAH – LANGKAH

Percobaan 1 : Proses induk (parent process) dan proses anak (child process)

1. Membuat proses sederhana. Simpan dengan nama `simpleFork.c`

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <sys/types.h>

int main(void)
{
    // process id
    int pid,i,endvalue;
    // use fork to create a new process
    endvalue=1000;
    printf("calling fork()\n");
    pid=fork();
    // check to see if fork worked
    if (pid<0)
        {
            printf("Fork failed\n");
            exit(0);
        }
    else if (pid ==0)
        {
             // in child process
             for (i=0; i<endvalue; i++)
                 {
                     printf("Child\n");
                     fflush(stdout);
                 }
        }
    else
        {
             // parent process
             wait(NULL);
             for(i=0; i<endvalue; i++)
                {
                     printf("Parent\n");
                     fflush(stdout);
                }
            printf("Child Complete\n");
            exit(0);
        }
    }
```

mengeluarkan program yang berjalan dari buffer ke terminalnya

Compile dengan perintah: `gcc simpleFork.c -Wall -o simpleFork`
Eksekusi dengan perintah: `./simpleFork`
Amati dan catat hasil yang ditampilkan.

2. Membuat proses induk (parent) dan anak (child). Simpan sebagai `child.c dan parent.c`

```
// Child Program
#include <stdio.h>
#include <unistd.h>

int main(void)
{
     while(1)
     {
       printf("Child");
       sleep(1);
     }
}
```

```
// Parent Program
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <error.h>
#include <signal.h>
#include <errno.h>

int main(void)
{
    pid_t pid;
    int status;
    if((pid =fork()) < 0)
    {
        // probably out of processes
        status =-1;
    }
    else if (pid == 0)
    {
        // in child so we execute process
        // use the execl function to to run a shell an execute the child program
        execl("/bin/sh","sh","-c","./child",(char *)0);
    }
    while(1)
    {
        sleep(1);
        printf("Parent");
    }
    printf("end of program");
}
```

Compile program child dengan perintah: `gcc child.c -Wall -o child`
Compile program parent dengan perintah: `gcc parent.c -Wall -o parent`
Eksekusi dengan perintah: `./parent`
Amati dan catat hasil yang ditampilkan.

3. Uji1.c (Menggunakaan execl)

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

int main ( )
{
execl ("/bin/ls", /* program yang dimuat - berikan secara full path */
      "ls", /* nama program yang akan dikirim ke argv[0] */
      "-l", /* parameter pertama (argv[1])*/
      "-a", /* parameter kedua (argv[2]) */
      NULL) ; /* terminasi arg list */
      printf ("EXEC Failed\n") ;
      /* Baris di atas akan dicetak saat terdapat kesalahan */
}
```

4. Uji2.c (Menggunakan execlp)

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

int main ( )
{
execlp ("ls", /* program yang dimuat - PATH dicari */
      "ls", /* nama program yang akan dikirim ke argv[0] */
      "-l", /* parameter pertama (argv[1])*/
      "-a", /* parameter kedua (argv[2]) */
      NULL) ; /* terminasi arg list */
      printf ("EXEC Failed\n") ;
      /* Baris di atas akan dicetak saat terdapat kesalahan */
}
```

5. Uji3.c (Menggunakan execv)

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

int main (argc, argv )
int argc ;
char *argv[ ] ;
{
      execv ("/bin/echo", /* program yang dimuat - hanya full path */
            &argv[0] ) ;
      printf ("EXEC Failed\n") ;
      /* Baris di atas akan dicetak saat terdapat kesalahan */
}
```

beda : pake pointer argumen buat ke fungsi

poin : pointre mau menjalankan array index ke berapa

7. Uji4.c (Menggunakan execvp)

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

int main (int argc, char *argv[] )
{
      execvp ("echo", /* program yang dimuat - PATH dicari */
      &argv[0] ) ;
      printf ("EXEC Failed\n") ;
      /* Baris di atas akan dicetak saat terdapat kesalahan */
}
```

8. Uji5.c (Menuliskan program ketika child dibuat untuk mengeksekusi sebuah *command*.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

int main ( )
{
      int forkresult ;

      printf ("%d: I am the parent. Remember my number!\n", getpid( ) ) ;
      printf ("%d: I am now going to fork ... \n", getpid( ) ) ;
      forkresult = fork ( ) ;
      if (forkresult != 0)
      { /* parent akan mengeksekusi kode ini */
            printf ("%d: My child's pid is %d\n", getpid ( ), forkresult ) ;
      }
      else /* forkresult == 0 */
      { /* child akan mengeksekusi kode ini */
            printf ("%d: Hi ! I am the child.\n", getpid ( ) ) ;
            printf ("%d: I'm now going to exec ls!\n\n\n", getpid ( ) ) ;
            execlp ("ls", "ls", NULL) ;
            printf ("%d: AAAAH ! ! My EXEC failed ! ! ! !\n", getpid ( ) ) ;
            exit (1) ;   program baru
      }
      printf ("%d: like father like son. \n", getpid ( ) ) ;
}
```

9. Uji6.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

int main ( )
{
      int number=0, statval; /* sinyal yang dikirim child ditangkap oleh
statval */

      printf ("%d: I'm the parent !\n", getpid ( ) ) ;
      printf ("%d: number = %d\n", getpid ( ), number ) ;
```

```
      printf ("%d: forking ! \n", getpid ( ) ) ;
      if ( fork ( ) == 0 )
      {
            printf ("%d: I'm the child !\n", getpid ( ) ) ;
            printf ("%d: number = %d\n", getpid ( ), number ) ;
            printf ("%d: Enter a number : ", getpid ( ) ) ;
            scanf ("%d", &number) ;
            printf ("%d: number = %d\n", getpid ( ), number ) ;
            printf ("%d: exiting with value %d\n", getpid ( ), number ) ;
            exit (number) ;
      }
      printf ("%d: number = %d\n", getpid ( ), number ) ;
      printf ("%d: waiting for my kid !\n", getpid ( ) ) ;
      wait (&statval) ;
      printf("statval = %d\n", statval);
      if ( WIFEXITED (statval) )
      {
            printf ("%d: my kid exited with status %d\n",
            getpid ( ), WEXITSTATUS (statval) ) ;
      }
      else
      {
            printf ("%d: My kid was killed off ! ! \n", getpid ( ) ) ;
      }
}
```

10. Uji7.c

Contoh *system call* wait(): program memunculkan dua child, lalu menunggu penyelesaian dari keduanya. Keduanya berperilaku berbeda-beda. Ubah/atur nilai sleep pada child untuk hasil pengamatan yang berbeda!

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

int main ( )
{
      int howmany, status, whichone, child1, child2 ;

      if ( (child1 = (int) fork()) == 0 ) /* Parent melahirkan child ke-1 */
      {
            printf("Hi, I am the first child, my ID is %d, and my parent ID
is %d\n", getpid(), getppid() ) ;
            sleep(10) ;
            printf("\nexiting first child\n");
            exit(0) ;
      }
      else if (child1 == -1)
      {
            perror("1st fork: something went wrong\n") ;
```

```
            exit(1) ;
      }

      if ( (child2 = (int) fork()) == 0 ) /* Parent melahirkan child ke-2 */
      {
            printf("Hi, I am the second child, my ID is %d, and my parent ID
is %d\n", getpid(), getppid() ) ;
            sleep(5) ;
            printf("\nexiting second child\n");
            exit(0) ;
      }
      else if (child2 == -1)
      {
            perror ("2nd fork: something went wrong\n") ;
            exit(1) ;
      }

      printf ("This is parent, my ID is %d\n", getpid()) ;
      howmany = 0 ;
      while (howmany < 2) /* Wait Twice */
      {
            whichone = (int) wait(&status) ;
            howmany++ ;
            printf("whichone id = %d\n", whichone);
            printf("child1 id = %d\n", child1);
            printf("child2 id = %d\n", child2);
            if (whichone == child1)
                  printf ("First child exited\n") ;
            else if (whichone == child2)
                  printf ("Second child exited\n") ;
            else
            {
                  printf ("not child exited\n");
                  printf ("whichone = %d\n", whichone);
            }
            if ( (status & 0xffff) == 0 )
                  printf ("correctly\n") ;
            else
                  printf ("uncorrectly\n") ;
      }
      printf ("\nParent terminated\n");
      return 0;
}
```

D. EVALUASI

1. Berdasarkan program **simpleFork.c,** jelaskan bagaimana proses mengetahui dirinya adalah induk atau anak?
2. Berdasarkan eksekusi **parent.c**, jelaskan tujuan pemanggilan fungsi `exec`.
3. Berikan kesimpulan cara penggunaan masing-masing keluarga exec berikut: execl, execlp, execv, execvp.
4. Apa fungsi argumen `NULL` pada argumen exec?
5. Amati **uji6.c** apakah PID proses berubah ketika exec dieksekusi?
6. Berdasarkan **uji6.c** apakah terbukti bahwa sinyal yang dikirim oleh exit() ditangkap oleh wait()? Berikan penjelasannya.
7. Berdasarkan **uji7.c** jelaskan siapa parent kedua child yang dibuat? Jelaskan pula bagaimana parent dapat mengenali child nya?