MODUL PRAKTIKUM SISTEM OPERASI

PRAKTIKUM III

MANAJEMEN PROSES DI LINUX

A. T U J U A N

1. Memahami perintah ps, jobs, bg, fg, top
2. Memahami perintah renice
3. Memahami perintah fork

B. LANGKAH – LANGKAH

Percobaan 1 : Status Proses

1. Melihat struktur Hirarki proses. Temukan nama proses yang pertama kali dibuat oleh linux.

   ```
   $ pstree
   ```

2. Instruksi ps (process status) digunakan untuk melihat proses pada sesi terminal yang aktif. PID adalah Nomor Identitas Proses, TTY adalah nama terminal dimana proses tersebut aktif, , COMMAND (CMD) merupakan instruksi yang digunakan.

   ```
   $ ps
   ```

3. ps x digunakan untuk melihat semua proses yang ada, termasuk yang tidak dalam kontrol terminal. Informasi state (STAT) berisi S (Sleeping) dan R (Running).

   ```
   $ ps x
   ```

4. Untuk melihat factor/elemen lainnya, gunakan option –u (user). %CPU adalah presentasi CPU time yang digunakan oleh proses tersebut, %MEM adalah presentasi system memori yang digunakan proses, SIZE adalah jumlah memori yang digunakan, RSS (*Real System Storage*) adalah jumlah memori yang digunakan, START adalah kapan proses tersebut diaktifkan

   ```
   $ ps -u
   ```

5. Mencari proses yang spesifik pemakai. Proses ini hanya terbatas pada proses milik pemakai, dimana pemakai tersebut melakukan login

   ```
   $ ps –u <user name>
   ```

6. Mencari proses lainnya gunakan option a (all) dan au (all user)

```
$ ps –a
$ ps –au
```

Percobaan 2 : foreground and background process

1. Menjalankan xlogo sebagai contoh proses yang terus aktif meng-update gambar ketika kita ubah ukuran jendelanya, dan tekan ctrl+C untuk keluar (dari xlogo).

   ```
   $ xlogo
   Ctrl + C
   ```

2. Memindahkan proses ke **background**

   `$ xlogo &` (catat id yang ditampilkan pada shell)

   `$ ps` (bandingkan id proses xlogo dengan id yang telah dicatat)

3. Melihat job yang aktif pada terminal

   `$ jobs` (perhatikan state dan apakah berada di foreground atau background)

4. Mengembalikan proses ke **foreground**

   ```
   $ jobs
   $ fg %<nomor proses>
   ```

5. Menghentikan proses

   ```
   $ xlogo
   Ctrl + Z
   ```
   (catat informasi yang ditampilkan)

   Cobalah mengubah ukuran jendela xlogo dalam dua keadaan: `running` dan `stopped`. Amati perbedaannya.

Percobaan 3: Menggunakan perintah top

1. Membuat beberapa proses yang bekerja dibelakang layar

```
$ man bash
CTRL+Z
$ ping 127.0.0.1 >> pinglokal &
$ ping google.com >> pingluar &
$ vi bgproc.sh
      #! /bin/bash
      i=0
      while [ true ];
      do
      i=$(($i+1))
      echo $i
      sleep 2
      done
$ chmod +x bgproc.sh
$ ./bgproc.sh
CTRL+Z
$ ps (apakah ada proses bgproc.sh?)
$ jobs (catat ID bgproc.sh)
```

2. Praktekkan langkah-langkah berikut :
   a) Masuk ke jendela top

      `$ top`

   b) Menampilkan menu bantuan

      `Tekan h atau ?`

   c) Menampilkan hanya 10 proses yang diamati

      `(dalam jendela top)`

      `Tekan n atau #`

      `10`

   d) Mengubah refresh rate menjadi 1 detik

      `$ top -d 1` jika pertama kali

      (atau tekan d , lalu masukkan waktunya)

   e) Mengubah refresh rate menjadi 10 detik

      `Tekan d`

      `10`

   f) Melakukan refresh sewaktu-waktu

      < tekan `space` atau `enter` ketika di dalam proses `top`>

   g) Menampilkan proses yang ada, diurutkan (sort) berdasarkan user

<tekan F atau O , lalu tekan e ketika di dalam proses top>

h) Menampilkan proses berdasarkan user tertentu

<tekan u, lalu ketikkan nama user>

i) Mengamati proses tertentu

$ ps (catat PID proses yang akan diamati)

$ top –p <PID proses yang akan diamati>

j) Keluar dari top

q

3. Matikan semua proses yang anda jalankan tadi (man bash, ping, bgproc.sh)

Percobaan 4: Nice dan Renice

1. Menjalankan perintah dengan nilai prioritas default

   $ ./bgproc.sh >> ./mydata.dat &

   $ ps (catat PID)

   $ jobs (periksa apakah sudah running)

2. Melihat nilai nice

   $ top –p <PID proses> (perhatikan nilai pada kolom NI)

   q (Keluar dari top)

3. Mengubah nilai prioritas proses

   $ renice –n 10 <PID proses> (catat pesan yang dihasilkan)

   $ renice –n 0 <PID proses>

   $ renice –n 19 <PID proses>

   $ renice –n 20 <PID proses> (catat pesan yang dihasilkan)

   $ renice –n -2 <PID proses> (catat pesan yang dihasilkan)

   $ sudo renice –n -5 <PID proses> (catat pesan yang dihasilkan)

4. Melakukan renice melalui top

   $ top

   r

   <PID yang akan kita renice>

   <nilai nice yang baru>

Percobaan 5: Pembuatan dan Eksekusi Proses

Pembuatan Banyak Proses

Fungsi **fork** untuk menciptakan proses baru. Jika operasi berhasil, maka kedua proses parent dan child dan keduanya melihat kembali fork, tetapi dengan nilai yang berbeda: mengembalikan nilai **0** dalam proses child dan mengembalikan ID proses child dalam proses parent. Jika pembuatan proses gagal, fork mengembalikan nilai **-1** dalam proses parent dan tidak ada child yang dibuat.

Atribut khusus proses child yang berbeda dari proses parent adalah:

- Proses child memiliki ID proses uniknya sendiri.
- ID proses parent dari proses child adalah ID proses dari proses parentnya. Proses child mendapatkan salinan sendiri dari pendeskripsi file terbuka proses parent. Selanjutnya mengubah atribut deskriptor file dalam proses parent tidak akan mempengaruhi deskriptor file dalam child, dan sebaliknya. Namun, kedua proses berbagi posisi file yang terkait dengan masing-masing deskriptor.
- Prosesor yang telah berlalu untuk proses child ditetapkan ke nol.
- Child tidak mewarisi kunci file yang ditetapkan oleh proses parent.
- Child tidak mewarisi alarm yang ditetapkan oleh proses parent.
- Kumpulan sinyal yang tertunda untuk proses child dihapus.

Lab1.c

```
#include <stdio.h>
#include <unistd.h> /* Berisi prototype fork*/

int main(void)
{
      printf("Hello World\n");
      fork();
      printf("<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<\n");
      printf("I am after forking\n");
      printf("\tI am process %d.\n", getpid());
      printf(">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>\n");
}
```

Ketika program ini dijalankan, pertama kali mencetak “Hello World!”. Ketika fork dijalankan, proses identik yang disebut child dibuat. Kemudian kedua proses parent dan proses child mulai dieksekusi pada pernyataan berikutnya. Perhatikan hal-hal berikut:

- Ketika sebuah fork dijalankan, segala sesuatu dalam proses parent disalin ke proses child. Ini termasuk nilai variabel, kode, dan deskriptor file.
- Mengikuti proses fork, proses child dan parent benar-benar independen.
- Tidak ada jaminan proses mana yang akan dicetak saya proses pertama.
- Proses child mulai eksekusi pada pernyataan segera setelah fork, bukan di awal program.
- Proses parent dapat dibedakan dari proses child dengan memeriksa nilai kembalinya panggilan fork. Fork mengembalikan nol ke proses child dan proses id dari proses child ke parent.
- Suatu proses dapat mengeksekusi banyak fork seperti yang diinginkan. Namun, waspada terhadap pengulangan fork yang tidak terbatas (ada jumlah maksimum proses yang diizinkan untuk satu pengguna).

Lab2.c

```
#include <stdio.h>
#include <unistd.h> /* Berisi prototype fork */

int main(void)
{
	int pid;

	printf("Hello World!\n");
	printf("I am the parent process and pid is : %d .\n", getpid());
	printf("Here i am before use of forking\n");
	pid = fork();
	printf("<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<\n");
	printf("Here I am just after forking\n");
	if (pid == 0)
		printf("I am the child process and pid is :%d.\n", getpid());
	else
		printf("I am the parent process and pid is: %d .\n", getpid());
	printf(">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>\n");
}
```

Lab3.c (Banyak Fork)

```
#include <stdio.h>
#include <unistd.h> /* Berisi prototype fork */

int main(void)
{
	printf("Here I am just before first forking statement\n");
	fork();
	printf("###############################################\n");
	printf("Here I am just after first forking statement\n");
	fork();
	printf("Here I am just after second forking statement\n");
	printf("\t\tHello World from process %d!\n", getpid());
}
```

Penyelesaian Proses

Fungsi yang dijelaskan dalam bagian ini digunakan untuk menunggu proses child untuk mengakhiri / menghentikan, dan menentukan statusnya. Fungsi-fungsi ini dideklarasikan di file header "sys / wait.h".
wait() akan memaksa proses parent untuk menunggu proses child berhenti. wait() mengembalikan pid dari child atau -1 untuk kesalahan. Status keluar child dikembalikan ke status_ptr.
exit() mengakhiri proses yang memanggil fungsi ini dan mengembalikan nilai return. Kedua program UNIX dan C (bercabang) dapat membaca nilai status.

Berdasarkan konvensi, **status 0** berarti penghentian normal. Nilai lainnya menunjukkan kesalahan atau kejadian yang tidak biasa. Banyak panggilan perpustakaan standar memiliki kesalahan yang ditentukan dalam file header sys / stat.h. Kita dapat dengan mudah mendapatkan konvensi kita sendiri.
Jika proses child harus dijamin untuk dieksekusi sebelum parent melanjutkan, panggilan sistem tunggu digunakan. Panggilan ke fungsi ini menyebabkan proses parent menunggu hingga salah satu proses childnya keluar. Panggilan **wait** mengembalikan proses id dari proses child, yang memberi parent kemampuan untuk menunggu proses child tertentu selesai.

Sleep

Suatu proses dapat menangguhkan untuk jangka waktu menggunakan perintah **sleep()**

Lab4.c: Menjamin proses child akan mencetak pesannya sebelum proses parent.

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h> /* mengandung fungsi wait */

void main(void)
{
	int pid;
	int status;

	printf("Hello World!\n");
	pid = fork();
	if (pid == -1) /* kondisi jika fork error */
	{
		perror("bad fork");
		exit(1);
	}
	if (pid == 0) printf("I am the child process.\n");
	else {
		wait(&status); /* parent menunggu child selesai */
		printf("I am the parent process.\n");
	}
}
```

Lab5.c

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

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();
	printf("################################\n");
	if (forkresult != 0) {
		/* proses parent akan mengeksekusi kode di bawah */
		printf("%d: My child's pid is %d\n", getpid(), forkresult);
		}
	else /* hasil fork == 0 */
		{ /* proses child akan mengeksekusi kode di bawah */
			printf("%d: Hi! I am the child.\n", getpid());
		}
	printf("%d: like father like son. \n", getpid());
}
```

Proses Orphan

Ketika parent terminate sebelum child-nya, child secara otomatis diadopsi oleh proses **"init"** asli yang **PID-nya** adalah **1**. Untuk mengilustrasikannya, masukkan statemen **sleep** ke kode child. Ini memastikan bahwa proses parent dihentikan sebelum child-nya.

Lab6.c

```
#include <stdio.h>

main()
{
	int pid ;

	printf("I'am the original process with PID %d and PPID %d.\n",
	getpid(), getppid()) ;
	pid = fork () ; /* Duplikasi proses, child dan parent */
	printf("################################################\n");
	if ( pid != 0 ) /* jika pid tidak nol, artinya saya parent*/
	{
		printf("I'am the parent with PID %d and PPID %d.\n",
		getpid(), getppid()) ;
		printf("My child's PID is %d\n", pid ) ;
	}
	else /* jika pid adalah nol, artinya saya child */
	{
		sleep(4); /* memastikan supaya parent lebih dulu di terminasi*/
		printf("I'm the child with PID %d and PPID %d.\n",
		getpid(), getppid()) ;
	}
	printf ("PID %d terminates.\n", getpid()) ;
}
```

Output:

I'am the original process with PID 5100 and PPID 5011.
I'am the parent process with PID 5100 and PPID 5011.
My child's PID is 5101
PID 5100 terminates. /* Parent terminasi */
I'am the child process with PID 5101 and PPID 1.
/* Orphaned (tidak memiliki parent/ yatimpiatu), yang menjadi proses parent adalah "init" dengan pid 1 */
PID 5101 terminates.

Proses Zombie

Suatu proses yang terminasi tidak dapat meninggalkan sistem sampai parent-nya menerima kode pengembaliannya. Jika proses parent nya sudah terminasi, itu sudah akan diadopsi oleh proses **"init"**, yang selalu menerima kode kembali child-nya. Namun, jika proses parent itu hidup tetapi tidak pernah mengeksekusi **wait ()**, kode pengembalian proses tidak akan pernah diterima dan proses akan tetap menjadi zombie.

Program berikut ini adalah membuat proses zombie, yang ditunjukkan dalam output dari utilitas ps. Ketika proses parent diterminasi, child diadopsi oleh **"init"** dan dapat terminasi dengan baik.

Lab7.c

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

int main ()
{
      int pid;

      pid = fork(); /* Duplikasi proses, Child dan parent */
      if (pid != 0) /* jika pid tidak nol, artinya saya parent */
      {
            while (1) /*Tidak Terminate dan tidak mengeksekusi wait()*/
                  sleep(100); /* berhenti selama 100 detik */
      }
      else /* pid adalah nol, artinya saya child */
      {
            exit(42); /* exit dengan angka berapapun */
      }
}
```

Output:

vlsi> a.out & /* Mengeksekusi program pada background */
[1] 5186
vlsi> ps /* Memperoleh status proses */
PID TT STAT TIME COMMAND
5187 p0 Z 0:00 <exiting> /* Proses anak zombie */
5149 p0 S 0:01 -csh (csh) /* Shell */
5186 p0 S 0:00 a.out /* Proses parent */
5188 p0 R 0:00 ps
vlsi> kill 5186 /* Terminasi proses parent */
[1] Terminated a.out
vlsi> ps /* Perhatikan bahwa zombie sudah hilang sekarang */
PID TT STAT TIME COMMAND
5149 p0 S 0:01 -csh (csh)
5189 p0 R 0:00 ps