LAPORAN PRAKTIKUM
PRAKTIKUM 3
MANAJEMEN PROSES DI LINUX

Disusun oleh:

Axel Anggian Hamonangan Purba

24060124140127

PRAKTIKUM SISTEM OPERASI

LAB A2

DEPARTEMEN INFORMATIKA

FAKULTAS SAINS DAN MATEMATIKA

UNIVERSITAS DIPONEGORO

2025

BAB I
PENDAHULUAN

1.1 Materi Praktikum

1.1. Lab1.c

#include <stdio.h>

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

int main(void)
{
    printf("Hello World\n");

    fork();  // Membuat proses baru (child)

    printf("<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<\n");
    printf("I am after forking\n");
    printf("\tI am process %d.\n", getpid());
    printf(">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>\n");

    return 0;
}
```

1.2. 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");

   return 0;
}
```

1.3. Lab3.c

```
#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());

    return 0;
}
```

1.4. Lab4.c

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

int 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");
    }

    return 0;
}
```

1.5. Lab5.c

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

int main(void)
{
    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 {
```

```
        /* proses child akan mengeksekusi kode di bawah */
        printf("%d: Hi! I am the child.\n", getpid());
    }

    printf("%d: like father like son.\n", getpid());

    return 0;
}
```

1.6. Lab6.c

```
#include <stdio.h>
#include <unistd.h>   /* getpid, getppid, fork, sleep */

int main(void)
{
    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) {
        /* proses 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 {
        /* proses child */
        sleep(4);   /* memastikan parent lebih dulu terminasi */
        printf("I'm the child with PID %d and PPID %d.\n",
               getpid(), getppid());
    }

    printf("PID %d terminates.\n", getpid());

    return 0;
}
```

1.7. Lab7.c

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

int main(void)
{
    int pid;

    pid = fork();  /* Duplikasi proses: child dan parent */

    if (pid != 0) {
        /* Parent process: tidak terminate dan tidak mengeksekusi wait() */
        while (1) {
            sleep(100);   /* berhenti selama 100 detik */
        }
    } else {
        /* Child process: langsung exit dengan status 42 */
        exit(42);
    }

    return 0;
}
```

2.1. Tujuan

2.1.1. Tujuan Praktikum

- Memahami konsep dasar proses dalam sistem operasi Linux, termasuk identitas proses (PID), status proses, serta hubungan hierarki antara parent process dan child process.
- Mampu menggunakan perintah-perintah manajemen proses seperti ps, jobs, bg, fg, dan pstree untuk mengamati, menampilkan, dan menganalisis proses yang sedang berjalan pada sistem.
- Mempelajari penggunaan perintah top untuk memantau penggunaan sumber daya seperti CPU, memori, serta mempelajari fitur-fitur pengurutan, penyaringan, dan pengaturan tampilan proses.
- Memahami mekanisme eksekusi proses dalam mode foreground dan background, serta mampu memindahkan proses antar mode menggunakan perintah shell.

-
- Mempelajari konsep prioritas proses dan pengaturan nilai nice, termasuk penggunaan perintah nice dan renice untuk mengubah prioritas eksekusi proses.
- Mengimplementasikan fungsi fork() dalam bahasa C untuk membuat proses baru dan memahami perbedaan antara proses parent dan child, termasuk bagaimana kedua proses berjalan secara bersamaan.
- Mengamati dan menganalisis fenomena proses khusus seperti proses orphan dan proses zombie, serta memahami bagaimana sistem menangani kondisi-kondisi tersebut.
- Menerapkan fungsi wait() dan sleep() untuk mengontrol eksekusi proses, sinkronisasi antara parent–child, dan mengatur alur perjalanan program.

BAB II
LANDASAN TEORI

2.1 Dasar Teori

Dalam lingkungan sistem operasi berbasis UNIX dan Linux, antarmuka command line (Command Line Interface/CLI) merupakan fondasi utama bagi pengguna untuk berinteraksi langsung dengan sistem. Melalui CLI, pengguna mengetikkan instruksi dalam bentuk baris perintah yang kemudian diproses oleh sebuah program khusus yang disebut shell. Pada sebagian besar distribusi Linux modern, shell yang digunakan adalah bash (Bourne Again Shell), versi pengembangan dari Bourne Shell yang menyediakan fitur lebih kaya, mulai dari scripting, manajemen proses, hingga ekspansi variabel.

Shell bertugas menerima input berupa teks, menganalisis sintaksnya, kemudian meneruskan perintah tersebut kepada kernel untuk dieksekusi. Selama proses berlangsung, hasil eksekusi biasanya ditampilkan kembali dalam bentuk teks ke layar terminal. CLI memberikan fleksibilitas dan kendali yang tinggi terhadap sistem, karena hampir seluruh operasi dapat dilakukan melalui perintah—termasuk mengelola file, menjalankan program, mengatur hak akses, serta melakukan pemantauan dan pengelolaan proses.

Dalam konteks manajemen proses, Linux menyediakan serangkaian instruksi bawaan untuk melihat status proses, memindahkan proses antara mode foreground dan background, mengatur prioritas proses, hingga membuat proses baru menggunakan fork(). Setiap proses memiliki process ID (PID) sebagai identitas unik. Informasi mengenai proses dapat diamati melalui perintah seperti ps, jobs, top, dan pstree, yang memberikan gambaran mengenai status, hierarki, konsumsi sumber daya, serta pengguna yang menjalankan proses tersebut.

Selain itu, sistem operasi Linux memiliki mekanisme multitasking yang memungkinkan banyak proses berjalan secara bersamaan. Untuk itu, konsep seperti nice value, renice, serta mode eksekusi foreground dan background menjadi penting. Proses yang dijalankan di background memungkinkan pengguna melanjutkan aktivitas di terminal tanpa menunggu proses tersebut selesai. Sebaliknya, proses foreground mengambil kendali penuh atas terminal hingga proses berakhir atau dihentikan.

Pada tingkat yang lebih rendah, pembuatan proses baru dalam Linux menggunakan panggilan sistem fork(), yang akan menduplikasi proses parent menjadi dua proses: parent dan child. Masing-masing proses kemudian berjalan secara independen. Selain fork, mekanisme wait() digunakan parent untuk menunggu child selesai, sehingga sinkronisasi proses dapat tercapai. Dari konsep ini muncul pula fenomena orphan (anak proses yang ditinggal mati oleh parent sehingga diadopsi oleh init/systemd) serta zombie (proses child yang telah selesai tetapi belum diambil statusnya oleh parent).

Melalui kombinasi CLI, shell, dan manajemen proses tersebut, Linux menyediakan kontrol mendalam terhadap cara kerja sistem, memungkinkan pengguna dan administrator untuk mengelola aplikasi, mendeteksi masalah performa, serta memahami interaksi antara proses di dalam sistem operasi.

BAB III
PEMBAHASAN

3.1. Pembahasan dan Penjelasan

3.1.1. Lab1.c

```
teaxelprb@LAPTOP-EUL8TGG2:~$ nano Lab1.c
axelprb@LAPTOP-EUL8TGG2:~$ gcc Lab1.c -o Lab1
axelprb@LAPTOP-EUL8TGG2:~$ ./Lab1
Hello World
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
I am after forking
        I am process 760.
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
I am after forking
        I am process 761.
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
```

Program ini menunjukkan bagaimana fungsi fork() membuat proses baru. Berikut alurnya:

- Program dimulai oleh satu proses (parent) dan mencetak:

  ```
  Hello World
  ```

- Saat fork() dipanggil, sistem membuat child process yang merupakan duplikasi dari parent.
- Setelah fork(), ada dua proses yang mengeksekusi baris-baris berikutnya secara independen:

  ```
  <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
  I am after forking
      I am process <PID>
  >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
  ```

- Karena prosesnya dua (parent dan child), blok output tersebut muncul dua kali, masing-masing dengan PID berbeda.
- Tidak ada jaminan proses mana yang mencetak lebih dulu; urutan bisa bergantian karena sistem menjadwalkan proses secara paralel.

3.1.2. Lab2.c

```
axelprb@LAPTOP-EUL8TGG2:~$ nano Lab2.c
axelprb@LAPTOP-EUL8TGG2:~$ gcc Lab2.c -o Lab2
axelprb@LAPTOP-EUL8TGG2:~$ ./Lab2
Hello World!
I am the parent process and pid is : 812 .
Here i am before use of forking
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Here I am just after forking
I am the parent process and pid is: 812 .
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Here I am just after forking
I am the child process and pid is :813.
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
```

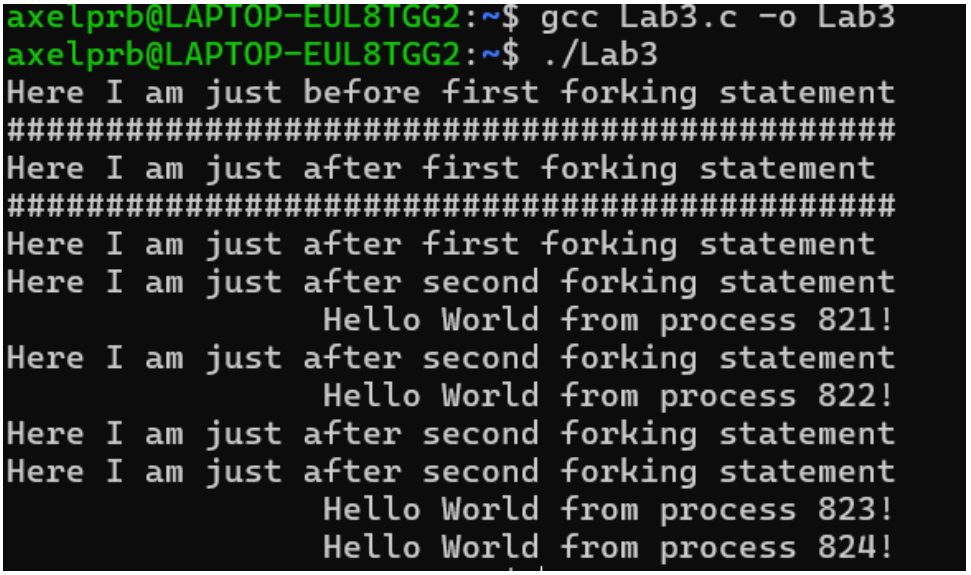
- Program dimulai oleh satu proses parent, mencetak:
  - "Hello World!"
  - PID parent
  - Pesan sebelum fork.
- `pid = fork();`
  - Membuat child process.
  - Pada child → `pid = 0`
  - Pada parent → `pid = <PID child>`
- Setelah `fork()`, dua proses menjalankan kode yang sama mulai dari baris berikutnya:

  ```
  <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
  Here I am just after forking
  ```
- Pengecekan kondisi:
- Jika `pid == 0` → child

  ```
  I am the child process and pid is: <PID child>
  ```
- Jika pid > 0 → parent

  ```
  I am the parent process and pid is: <PID
  parent>
  ```
- Menjelaskan bagaimana membedakan parent dan child setelah pemanggilan `fork()`.
- Kedua proses tetap menjalankan kode yang sama, tetapi kondisi `pid == 0` digunakan untuk menentukan siapa child dan siapa parent.

3.1.3. Lab3.c



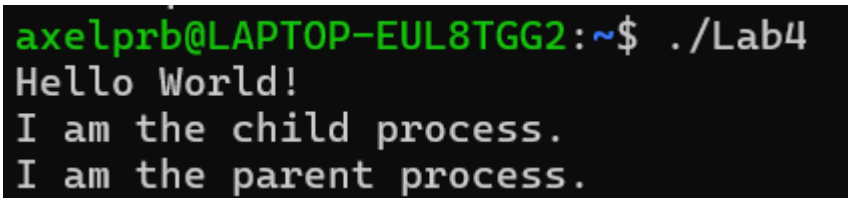
- Sebelum fork pertama — hanya 1 proses
- Program dimulai oleh satu proses parent, mencetak:
  `Here I am just before first forking`
- Setelah fork pertama → menjadi 2 proses
- `fork()`; pertama membuat 1 child, sehingga total proses:
  - Parent
  - Child
  - Kedua proses ini mengeksekusi baris berikutnya bersama-sama, sehingga output berikut dicetak 2 kali:

  `################################################`
  `Here I am just after first forking`
- Fork kedua dijalankan oleh ke-2 proses tadi
  - Kedua proses hasil fork pertama masing-masing memanggil `fork()` lagi:
  - Parent menghasilkan child baru
  - Child juga menghasilkan child baru
  - Total proses setelah fork kedua = 4 proses (karena setiap proses membuat satu proses baru)
- Baris setelah fork kedua dicetak 4 kali
  - Empat proses ini menjalankan baris berikut:
    `Here I am just after second forking`

`Hello World from process <PID>`

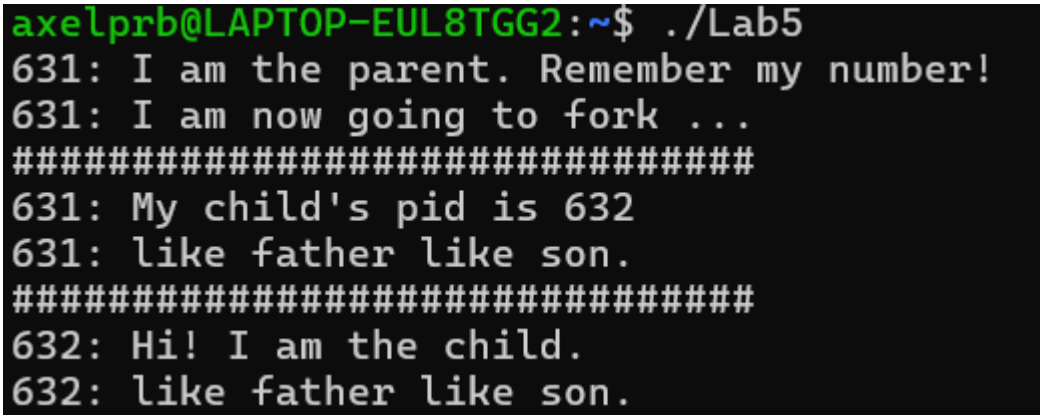
- Setiap proses memiliki PID yang berbeda, sehingga output muncul empat kali.

3.1.4. Lab4.c



- Program dimulai oleh proses parent
  - Mencetak:
    `Here I am before forking`
  - Masih 1 proses.
- `fork()` dijalankan → menjadi dua proses
  - Parent (pid > 0)
  - Child (pid == 0)
- Pemakaian `sleep(1)` pada child
  - Child menunggu 1 detik.
  - Parent tidak sleep, sehingga parent mencetak duluan:
    `I am the parent, pid = …`
- Setelah parent selesai, barulah child bangun dan mencetak:
  `I am the child, pid = ...`
  - Dengan demikian, urutan output bisa dikontrol.
- Keduanya mencetak baris terakhir
  - Karena kedua proses mencapai:
    `End of process <PID>`
  - Maka output ini muncul dua kali, masing-masing dengan PID berbeda.

3.1.5. Lab5.c



- Program dimulai oleh parent
  - Mencetak PID parent dua kali:

    `<PID>: I am the parent. Remember my number!`

    `<PID>: I am now going to fork …`
  - Masih satu proses.
- `forkresult = fork();` dijalankan
  - Setelah fork, muncul dua proses:
  - Parent: `forkresult = PID_child`
  - Child: `forkresult = 0`
  - Keduanya mengeksekusi baris berikutnya.
- Keduanya mencetak garis pemisah
  - Karena dua proses kini berjalan:

    `################################`

    `################################`
- Percabangan if `(forkresult != 0)`
  - Parent `(forkresult != 0)`
  - Mencetak:

    `<PID_parent>: My child's pid is <PID_child>`

    `Child (forkresult == 0)`
  - Mencetak:

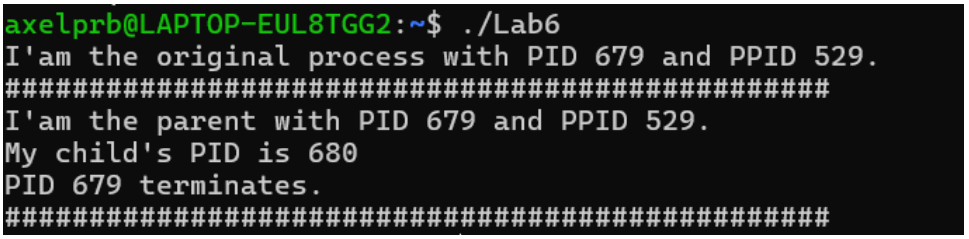
    `<PID_child>: Hi! I am the child.`

- Keduanya mengeksekusi baris terakhir

  `<PID>: like father like son.`

  - Dicetak dua kali (parent dan child).
- Menunjukkan perbedaan perilaku parent dan child setelah fork().
- Variabel forkresult digunakan untuk membedakan proses.
- Kedua proses tetap menjalankan kode setelah percabangan.
- Output tampil dua kali untuk baris yang tidak ada dalam blok if/else.

3.1.6. Lab6.c



Lab6.c digunakan untuk menunjukkan bagaimana proses child menjadi orphan ketika parent-nya selesai lebih dulu.

- Program dimulai oleh satu proses parent, yang menampilkan PID dan PPID awal.
- Ketika fork() dipanggil, sistem membuat child process.
- Parent: pid != 0
- Child : pid == 0
- Proses parent mencetak informasi dirinya dan PID child, lalu langsung melakukan terminasi.
- Hal ini menyebabkan child kehilangan parent.
- Child menjalankan sleep(2) agar parent mati lebih dulu dan statusnya berubah menjadi orphan.
- Setelah bangun dari sleep, child mencetak PID dan PPID barunya.
- PPID akan menjadi 1 (init atau systemd), menunjukkan bahwa child telah diadopsi oleh proses init.
- Child kemudian mencetak pesan terakhir dan terminasi.

3.1.7. Lab7.c

```
axelprb@LAPTOP-EUL8TGG2:~$ nano Lab7.c
axelprb@LAPTOP-EUL8TGG2:~$ gcc Lab7.c -o Lab7
axelprb@LAPTOP-EUL8TGG2:~$ ./Lab7
^C
```

- Program melakukan `fork()`
    - Setelah `fork()`, terbentuk dua proses:
    - Parent → nilai `pid != 0`
    - Child → nilai `pid == 0`
- Proses Child langsung selesai
    - Child mengeksekusi:
    - `exit(42);`
    - Artinya:
    - Child berhenti secara normal
    - Sistem menyimpan status keluaran child agar parent bisa mengambilnya melalui `wait()`
    - Namun pada Lab7.c, parent tidak pernah memanggil `wait()`, sehingga status child tidak pernah diambil.
- Parent tidak pernah mati dan tidak memanggil `wait()`
    - Parent masuk ke loop:
    - `while (1)`
    - `    sleep(100);`
    - Ini membuat parent:
    - Tetap hidup
    - Tidak pernah memanggil wait()
    - Tidak pernah mengambil status child
    - Akibatnya, child tetap berada di tabel proses dalam kondisi zombie.
- Zombie terlihat di ps
    - Jika menjalankan:
    - `ps`
    - akan terlihat child dengan:
    - STAT: Z (Zombie)

        - COMMAND: <defunct> atau <exiting>
        - Proses zombie tidak berjalan dan tidak memakai CPU, tetapi masih menempati entri proses.
- Zombie baru hilang setelah parent mati
    - Jika parent dihentikan:

      `kill <PID_parent>`
    - maka:
        - Zombie akan diadopsi oleh init/systemd (PID 1)
        - init akan memanggil wait() otomatis
        - Zombie hilang dari tabel proses

BAB IV
PENUTUP

4.1. Kesimpulan

Melalui praktikum manajemen proses di Linux, dapat dipahami bahwa sistem operasi menyediakan berbagai mekanisme untuk membuat, mengelola, memantau, dan mengakhiri proses. Perintah seperti ps, jobs, fg, bg, dan top memungkinkan pengguna melihat status proses, pergerakan prioritas, serta interaksi antara proses foreground dan background. Konsep prioritas proses juga dipelajari melalui penggunaan nice dan renice, yang menunjukkan bagaimana nilai prioritas memengaruhi kesempatan eksekusi suatu proses.

Selain itu, percobaan menggunakan fungsi fork() menjelaskan bagaimana proses baru diciptakan melalui duplikasi proses parent, serta bagaimana kedua proses tersebut dapat berjalan paralel. Melalui percobaan lanjut seperti penggunaan wait(), sleep(), dan pengamatan pada kondisi khusus seperti proses orphan dan zombie, terlihat bahwa interaksi antara parent dan child sangat menentukan alur hidup suatu proses dalam sistem. Dengan demikian, praktikum ini memberikan pemahaman menyeluruh mengenai perilaku dasar proses di Linux, termasuk penciptaan, penjadwalan, sinkronisasi, dan terminasi proses.