# LAPORAN PRAKTIKUM
# PRAKTIKUM 4
# PEMBUATAN PROSES II


**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

### 1.1.1. Uji1.c

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 */
}
```

### 1.1.2. Uji2.c

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 */
}
```

### 1.1.3. Uji3.c

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 */
}
```

### 1.1.4. Uji4.c

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 */
}
```

### 1.1.5. Uji5.c

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

### 1.1.6. Uji6.c

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 ( ) ) ;
      }
}
```

### 1.1.7. Uji7.c

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;
}
```

# BAB II
# LANDASAN TEORI

## 2.1 Dasar Teori

Dalam sistem operasi berbasis UNIX dan Linux, konsep proses dan mekanisme pembuatannya merupakan salah satu fondasi terpenting dalam pengelolaan eksekusi program. Setiap program yang berjalan pada sistem direpresentasikan sebagai sebuah proses—entitas yang memiliki ruang memori sendiri, konteks eksekusi, serta identitas berupa Process ID (PID). Dalam praktik sehari-hari, proses dapat saling berinteraksi, saling menunggu, atau bahkan menciptakan proses baru untuk melakukan tugas tertentu secara paralel.

Mekanisme utama pembentukan proses baru di lingkungan Linux dilakukan melalui pemanggilan system call fork(). Fungsi ini menduplikasi proses yang sedang berjalan (parent) dan menghasilkan sebuah proses baru (child) yang merupakan salinan hampir sempurna dari induknya. Setelah fork() dipanggil, eksekusi program bercabang menjadi dua jalur: satu untuk proses parent dan satu lagi untuk proses child. Kedua proses akan menjalankan instruksi selanjutnya secara independen, namun mereka dapat dibedakan melalui nilai kembalian fork(): proses child menerima nilai 0, sedangkan parent menerima PID dari proses child yang baru dibuat. Perbedaan nilai inilah yang memungkinkan program menentukan blok kode mana yang harus dijalankan oleh masing-masing proses.

Salinan lingkungan yang diwarisi child dari parent meliputi data segment, heap, stack, serta hampir seluruh konteks eksekusi. Dengan demikian, child akan memiliki kondisi awal yang sama seperti induknya pada saat fork() dipanggil. Setelah proses child terbentuk, ia dapat memodifikasi lingkungannya atau bahkan mengganti program yang sedang berjalan menggunakan keluarga fungsi exec. Keluarga system call ini—seperti execl, execlp, execv, dan execvp—bertugas memuat program baru ke dalam ruang memori proses sehingga menggantikan program sebelumnya sepenuhnya. Perbedaan utama dari fungsi-fungsi tersebut terletak pada cara pemberian argumen serta penentuan lokasi program (apakah harus full path atau dapat dicari melalui variabel PATH).

Sementara fork() menciptakan proses baru, sinkronisasi antara parent dan child sering dilakukan menggunakan system call wait() atau waitpid(). Fungsi ini memungkinkan parent menunggu hingga child selesai dieksekusi. Ketika child keluar melalui exit(), ia mengirimkan suatu status yang kemudian dapat diterima oleh parent melalui wait(). Mekanisme ini sangat penting untuk mencegah munculnya proses zombie—yaitu proses child yang telah selesai tetapi statusnya belum diambil oleh parent. Sebaliknya, jika parent mati lebih dahulu dan tidak melakukan wait(), child berpotensi menjadi proses orphan dan kemudian diadopsi oleh proses init/systemd.

Dalam konteks praktikum, pemahaman terhadap interaksi antara fork(), keluarga exec, serta wait() menjadi kunci untuk memahami bagaimana Linux mengelola multitasking dan manajemen proses tingkat rendah. Dengan mempelajari bagaimana parent dan child terbentuk, bagaimana mereka berkomunikasi melalui nilai kembalian dan status exit, serta bagaimana sebuah proses dapat mengganti dirinya sendiri dengan program lain, kita dapat melihat gambaran utuh bagaimana sistem operasi bekerja pada lapisan paling dasar.

Keseluruhan mekanisme ini memberikan fleksibilitas bagi programmer maupun administrator sistem untuk mengelola proses secara efisien, menjalankan program secara paralel, mengatur alur eksekusi, dan memastikan setiap proses berjalan sesuai dengan kontrol yang diinginkan.

# BAB III
# PEMBAHASAN

## 3.1. Pembahasan dan Penjelasan

### 3.1.1. Uji1.c

```
axelprb@LAPTOP-EUL8TGG2:~$ nano Uji1.c
axelprb@LAPTOP-EUL8TGG2:~$ gcc Uji1.c -o Uji1
axelprb@LAPTOP-EUL8TGG2:~$ ./Uji1
total 284
drwxr-x--- 6 axelprb axelprb  4096 Dec  3 18:20 .
drwxr-xr-x 3 root    root     4096 Nov 12 00:00 ..
-rw------- 1 axelprb axelprb  3864 Dec  3 13:46 .bash_history
-rw-r--r-- 1 axelprb axelprb   220 Nov 12 00:00 .bash_logout
-rw-r--r-- 1 axelprb axelprb  3860 Nov 29 12:47 .bashrc
drwx------ 4 axelprb axelprb  4096 Nov 29 12:48 .cache
-rw------- 1 axelprb axelprb    20 Nov 12 00:05 .lesshst
drwxr-xr-x 5 axelprb axelprb  4096 Nov 29 12:47 .local
-rw-rw-r-- 1 axelprb axelprb     0 Dec  3 13:24 .motd_shown
-rw-r--r-- 1 axelprb axelprb   896 Nov 29 12:47 .profile
-rw-r--r-- 1 axelprb axelprb     0 Nov 12 00:03 .sudo_as_admin_successful
-rw------- 1 axelprb axelprb  7122 Dec  3 13:27 .viminfo
-rwxr-xr-x 1 axelprb axelprb 16088 Nov 26 09:10 Child
-rw-r--r-- 1 axelprb axelprb   159 Nov 26 09:19 Child.c
-rwxr-xr-x 1 axelprb axelprb 16088 Dec  2 21:49 Lab1
-rw-r--r-- 1 axelprb axelprb   307 Dec  2 21:48 Lab1.c
-rwxr-xr-x 1 axelprb axelprb 16088 Dec  2 22:07 Lab2
-rw-r--r-- 1 axelprb axelprb   585 Dec  2 22:09 Lab2.c
-rwxr-xr-x 1 axelprb axelprb 16088 Dec  2 22:09 Lab3
-rw-r--r-- 1 axelprb axelprb   426 Dec  2 22:09 Lab3.c
-rwxr-xr-x 1 axelprb axelprb 16176 Dec  3 13:29 Lab4
-rw-r--r-- 1 axelprb axelprb   483 Dec  3 13:28 Lab4.c
-rwxr-xr-x 1 axelprb axelprb 16088 Dec  3 13:30 Lab5
-rw-r--r-- 1 axelprb axelprb   609 Dec  3 13:30 Lab5.c
-rwxr-xr-x 1 axelprb axelprb 16176 Dec  3 13:31 Lab6
-rw-r--r-- 1 axelprb axelprb   778 Dec  3 13:31 Lab6.c
-rwxr-xr-x 1 axelprb axelprb 16040 Dec  3 13:32 Lab7
-rw-r--r-- 1 axelprb axelprb   465 Dec  3 13:33 Lab7.c
-rwxr-xr-x 1 axelprb axelprb 16088 Nov 26 09:28 Parent
-rw-r--r-- 1 axelprb axelprb   606 Nov 26 09:28 Parent.c
-rwxr-xr-x 1 axelprb axelprb 16000 Dec  3 18:20 Uji1
-rw-r--r-- 1 axelprb axelprb   478 Dec  3 18:20 Uji1.c
-rwxr-xr-x 1 axelprb axelprb    74 Nov 19 09:00 danusan.sh
-rw-r--r-- 1 axelprb axelprb   199 Nov 15 17:24 evaluasi
-rw-r--r-- 1 axelprb axelprb    54 Nov 15 17:37 myLinux
drwxr-xr-x 2 axelprb axelprb  4096 Nov 15 18:29 mylinux
-rwxr-xr-x 1 axelprb axelprb 16176 Nov 26 09:06 simplefork
-rw-r--r-- 1 axelprb axelprb   745 Nov 26 09:02 simplefork.c
drwxr-xr-x 2 axelprb axelprb  4096 Nov 15 18:29 thelinux
```

Program ini menunjukkan bagaimana `fork()` menciptakan dua proses: parent dan child. Setelah `fork()` dipanggil, kedua proses menjalankan kode yang sama tetapi dibedakan melalui nilai kembalian `fork()`. Child mendapatkan nilai 0 sehingga menjalankan loop yang mencetak "Child" sebanyak 1000 kali. Parent menerima PID child dan menjalankan wait(NULL) untuk menunggu child selesai sebelum mencetak "Parent" berulang. Percobaan ini memperlihatkan identifikasi induk–anak serta sinkronisasi sederhana menggunakan wait().

### 3.1.2. Uji2.c

```
axelprb@LAPTOP-EUL8TGG2:~$ ./Uji2
total 304
drwxr-x--- 6 axelprb axelprb  4096 Dec  3 18:22 .
drwxr-xr-x 3 root    root     4096 Nov 12 00:00 ..
-rw------- 1 axelprb axelprb  3864 Dec  3 13:46 .bash_history
-rw-r--r-- 1 axelprb axelprb   220 Nov 12 00:00 .bash_logout
-rw-r--r-- 1 axelprb axelprb  3860 Nov 29 12:47 .bashrc
drwx------ 4 axelprb axelprb  4096 Nov 29 12:48 .cache
-rw------- 1 axelprb axelprb    20 Nov 12 00:05 .lesshst
drwxr-xr-x 5 axelprb axelprb  4096 Nov 29 12:47 .local
-rw-rw-r-- 1 axelprb axelprb     0 Dec  3 13:24 .motd_shown
-rw-r--r-- 1 axelprb axelprb   896 Nov 29 12:47 .profile
-rw-r--r-- 1 axelprb axelprb     0 Nov 12 00:03 .sudo_as_admin_successful
-rw------- 1 axelprb axelprb  7122 Dec  3 13:27 .viminfo
-rwxr-xr-x 1 axelprb axelprb 16088 Nov 26 09:10 Child
-rw-r--r-- 1 axelprb axelprb   159 Nov 26 09:19 Child.c
-rwxr-xr-x 1 axelprb axelprb 16088 Dec  2 21:49 Lab1
-rw-r--r-- 1 axelprb axelprb   307 Dec  2 21:48 Lab1.c
-rwxr-xr-x 1 axelprb axelprb 16088 Dec  2 22:07 Lab2
-rw-r--r-- 1 axelprb axelprb   585 Dec  2 22:09 Lab2.c
-rwxr-xr-x 1 axelprb axelprb 16088 Dec  2 22:09 Lab3
-rw-r--r-- 1 axelprb axelprb   426 Dec  2 22:09 Lab3.c
-rwxr-xr-x 1 axelprb axelprb 16176 Dec  3 13:29 Lab4
-rw-r--r-- 1 axelprb axelprb   483 Dec  3 13:28 Lab4.c
-rwxr-xr-x 1 axelprb axelprb 16088 Dec  3 13:30 Lab5
-rw-r--r-- 1 axelprb axelprb   609 Dec  3 13:30 Lab5.c
-rwxr-xr-x 1 axelprb axelprb 16176 Dec  3 13:31 Lab6
-rw-r--r-- 1 axelprb axelprb   778 Dec  3 13:31 Lab6.c
-rwxr-xr-x 1 axelprb axelprb 16040 Dec  3 13:32 Lab7
-rw-r--r-- 1 axelprb axelprb   465 Dec  3 13:33 Lab7.c
-rwxr-xr-x 1 axelprb axelprb 16088 Nov 26 09:28 Parent
-rw-r--r-- 1 axelprb axelprb   606 Nov 26 09:28 Parent.c
-rwxr-xr-x 1 axelprb axelprb 16000 Dec  3 18:20 Uji1
-rw-r--r-- 1 axelprb axelprb   478 Dec  3 18:20 Uji1.c
-rwxr-xr-x 1 axelprb axelprb 16000 Dec  3 18:22 Uji2
-rw-r--r-- 1 axelprb axelprb   424 Dec  3 18:21 Uji2.c
-rwxr-xr-x 1 axelprb axelprb    74 Nov 19 09:00 danusan.sh
-rw-r--r-- 1 axelprb axelprb   199 Nov 15 17:24 evaluasi
-rw-r--r-- 1 axelprb axelprb    54 Nov 15 17:37 myLinux
drwxr-xr-x 2 axelprb axelprb  4096 Nov 15 18:29 mylinux
-rwxr-xr-x 1 axelprb axelprb 16176 Nov 26 09:06 simplefork
-rw-r--r-- 1 axelprb axelprb   745 Nov 26 09:02 simplefork.c
drwxr-xr-x 2 axelprb axelprb  4096 Nov 15 18:29 thelinux
```

Pada percobaan ini parent melakukan `fork()`. Jika parent, ia terus mencetak "Parent". Jika child, program child tidak menjalankan kode C lainnya tetapi langsung diganti dengan eksekusi file child melalui:

```
execl("/bin/sh", "sh", "-c", "./child", NULL);
```

Perintah ini memerintahkan shell untuk menjalankan program child. Program child mencetak "Child" terus-menerus. Percobaan ini menunjukkan bagaimana parent dapat membuat child yang menjalankan program lain menggunakan execl, serta bagaimana dua program berjalan paralel.

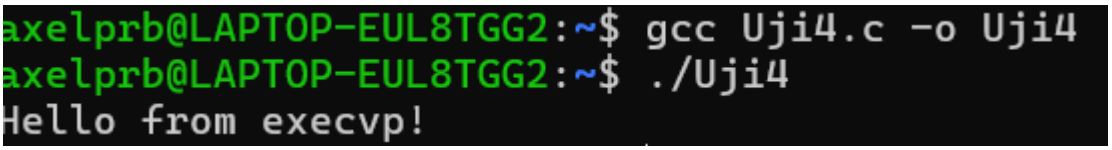
### 3.1.3. Uji3.c

`execl()` digunakan untuk mengeksekusi program baru dengan path lengkap program. Pada Uji3, program memanggil:

```
execl("/bin/ls", "ls", "-l", "-a", NULL);
```

Hal ini menggantikan proses yang sedang berjalan dengan program ls lengkap dengan argumen -l dan -a. Jika execl berhasil, baris setelahnya tidak akan dieksekusi. Jika gagal, baris "EXEC Failed" akan muncul. Intinya, execl membutuhkan full path program dan argumen dituliskan satu per satu.

### 3.1.4. Uji4.c

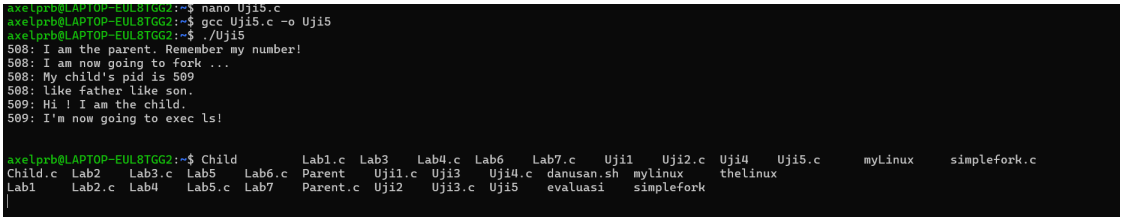



`execlp()` mirip dengan `execl()`, tetapi tidak memerlukan path lengkap. Program akan mencari executable berdasarkan variable lingkungan PATH. Pada Uji4, pemanggilan:

```
execlp("ls", "ls", "-l", "-a", NULL);
```

memungkinkan program menemukan ls secara otomatis. Keunggulannya adalah lebih fleksibel, tidak perlu menuliskan path lengkap, dan tetap menggantikan proses dengan program baru.

### 3.1.5. Uji5.c

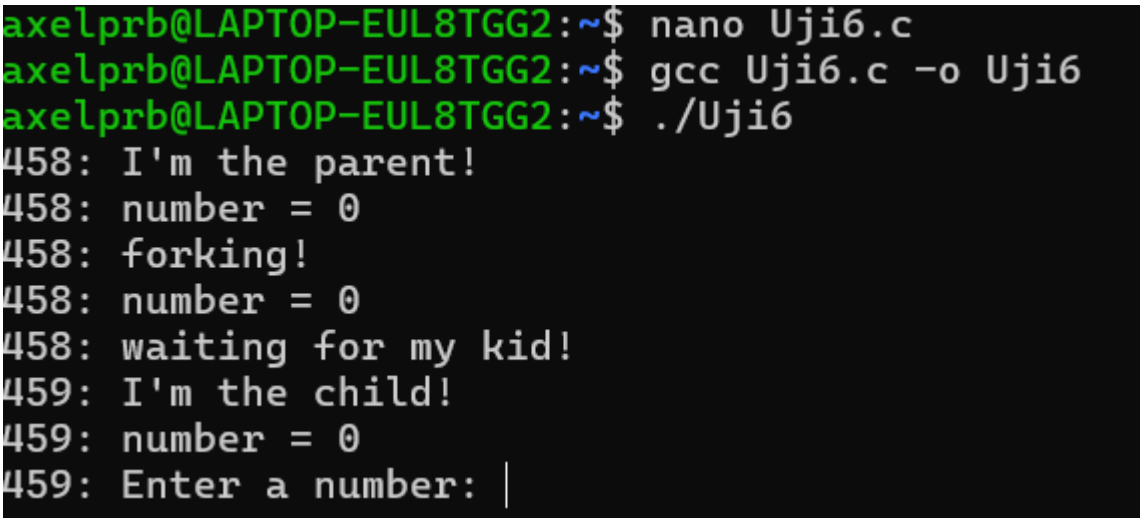



`execv()` menerima argumen dalam bentuk array string `(argv)`. Pada Uji5:

```
execv("/bin/echo", &argv[0]);
```

Program akan diganti dengan program echo yang dipanggil melalui path lengkap. Karena execv memakai array, argumen yang diteruskan harus disusun dalam bentuk list/array. Jika berhasil, program utama berhenti dan hanya program echo yang berjalan.

### 3.1.6. Uji6.c



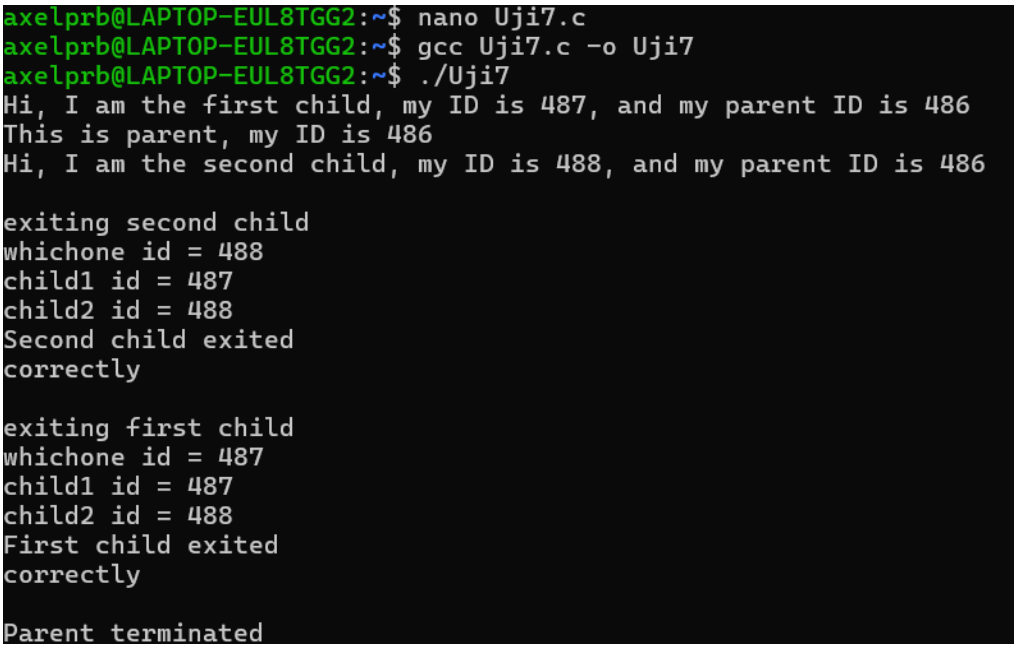

Pada Uji6, proses parent membuat child menggunakan `fork()`. Child meminta input angka dari pengguna, lalu keluar menggunakan `exit(number)`, di mana number adalah nilai yang dimasukkan pengguna.
Parent kemudian menunggu child dengan:

```
wait(&statval);
```

statval menangkap status exit child. Parent dapat mengecek:
Apakah child keluar secara normal `(WIFEXITED)`
Berapa nilai exit dari child `(WEXITSTATUS)`

Percobaan ini membuktikan bahwa nilai `exit()` benar-benar dikirimkan dan diterima oleh parent melalui `wait()`. Selain itu, PID proses tidak berubah setelah exec (karena exec tidak digunakan dalam percobaan ini), dan hanya child yang melakukan exit sesuai nilai input.

### 3.1.7. Uji7.c




Uji7 memperlihatkan bagaimana parent membuat dua child secara berurutan menggunakan dua pemanggilan `fork()`. Masing-masing child melakukan pekerjaan berbeda dan keluar pada waktu yang berbeda (karena sleep berbeda).

Parent kemudian:

Melakukan `wait()` dua kali dalam loop

Menerima PID child yang selesai terlebih dahulu (urutan tidak ditentukan, tergantung sleep)

Membandingkan PID yang dikembalikan dengan PID child1 dan child2

Dengan membandingkan PID, parent dapat mengetahui child mana yang selesai lebih dahulu. Percobaan ini juga menunjukkan bahwa proses parent tetap sama untuk kedua child, dan mekanisme `wait()` digunakan untuk sinkronisasi multi-child.

# BAB IV
# PENUTUP

## 4.1. Kesimpulan

Praktikum ini memberikan pemahaman mendalam mengenai cara kerja proses pada sistem operasi Linux, khususnya terkait pembuatan proses baru dengan `fork()`, penggantian program dengan keluarga exec, serta mekanisme sinkronisasi menggunakan `wait()`. Melalui serangkaian uji, terlihat bahwa:

- `fork()` membagi eksekusi menjadi dua jalur—parent dan child—yang dapat dibedakan dari nilai kembalian fungsi tersebut.
- `exec()` memungkinkan sebuah proses menggantikan dirinya dengan program lain, tanpa membuat proses baru, sehingga PID tetap sama.
- Berbagai variasi `exec (execl, execlp, execv, execvp)` memberi fleksibilitas dalam pemanggilan program, baik melalui path lengkap maupun pencarian berdasarkan PATH.
- `wait()` dan nilai exit child menunjukkan bahwa parent dapat menerima status keluaran child, mencegah zombie process, serta memastikan sinkronisasi antara parent dan child.
- Percobaan multi-child memperlihatkan bagaimana parent mengenali proses mana yang selesai terlebih dahulu melalui PID yang dikembalikan oleh wait().

Secara keseluruhan, praktikum ini memperlihatkan dasar-dasar penting manajemen proses di Linux: bagaimana proses dibuat, dijalankan, diganti, dan dikelola. Pemahaman ini menjadi fondasi dalam pemrograman sistem dan pengelolaan aplikasi yang berjalan secara paralel di lingkungan UNIX/Linux.