Sistem Operasi : Proses pada LINUX

Moh. Fajar
fajarmks@yahoo.com

Abstract :

Tulisan ini berusaha mengupas tentang konsep dasar proses pada sistem operasi khususnya sistem operasi linux seperti identifikasi proses, pembuatan proses dan komunikasi proses menggunakan signal. Setelah membaca tulisan ini diharapkan pembaca memiliki gambaran yang cukup jelas tentang konsep dasar proses pada sistem operasi linux.

1. Pendahuluan

Proses merupakan konsep pokok pada sistem operasi, karena salah satu tugas utama sistem operasi adalah bagaimana mengatur proses – proses yang berjalan di sistem. Sebenarnya apakah Proses itu?
Berikut beberapa definisi proses pada sistem operasi adalah :

1. Program yang sedang dalam keadaan dieksekusi.
2. Unit kerja terkecil yang secara individu memiliki sumber daya dan dijadwalkan oleh sistem operasi.

Sistem operasi mengolah seluruh proses yang ada di sistem dan bertugas mengalokasikan sumber daya – sumber daya ke proses yang membutuhkan sesuai dengan kebijaksanaan tertentu. Sumber daya yang dibutuhkan proses diantaranya CPU, memori, file serta I/O device.

2. Keadaan Proses

Proses – proses yang dikelola oleh sistem operasi akan melalui serangkaian keadaan yang merupakan bagian dari aktivitasnya. Keadaan proses ini disebut sebagai status proses yang terdiri dari:

Status New yaitu status dimana proses sedang dibuat.

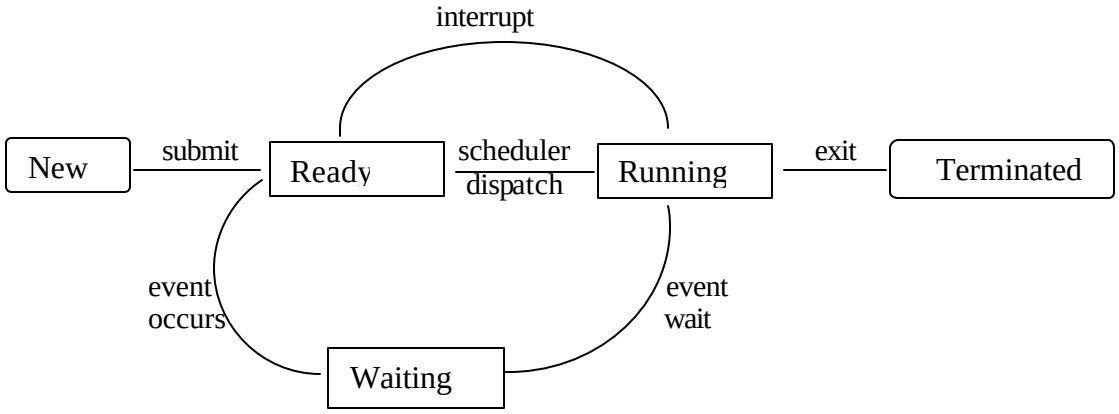
Status Ready yaitu status dimana proses siap dieksekusi tetapi CPU belum tersedia karena sedang mengerjakan proses lain.

Status Waiting yaitu status dimana proses sedang menunggu suatu kejadian tertentu. Misalnya sedang menunggu operasi I/O selesai, menunggu signal dari proses lain, tersedianya memori, dsb.

Status Running yaitu status dimana proses dieksekusi. Pada status ini CPU sedang mengeksekusi instruksi – instruksi pada proses.

Status Terminated yaitu status dimana proses diakhiri.

Kelima status proses tersebut dapat digambarkan pada diagram berikut:



"Diagram status proses"

- Sebuah proses menjadi Waiting karena proses tersebut menunggu suatu kejadian tertentu seperti selesainya operasi I/O, misalnya perekaman data ke disk karena pada saat perekaman dilakukan proses sedang tidak menggunakan CPU maka scheduler segera mengalokasikan CPU ke proses lain yang telah Ready. Apabila kejadian yang ditunggu telah selesai maka proses dipindahkan kembali ke antrian Ready dan siap dijadwalkan.

- Sebuah proses dari keadaan Running dapat menjadi Ready kembali karena diinterupsi oleh proses lain. Interupsi dapat disebabkan karena jatah waktu yang diberikan CPU ke proses tersebut telah habis sementara proses masih memerlukan sejumlah waktu untuk selesai. Jatah waktu yang diberikan sering disebut sebagai quantum time (time slice) yang dapat berkisar antara 1 hingga 100 milidetik. Interupsi suatu proses terkait erat dengan strategi penjadwalan proses yang digunakan sistem operasi yaitu strategi preemptive dimana suatu proses dapat saja disela oleh proses lain pada saat Running.
- Sebuah proses menjadi Terminated disebabkan oleh beberapa hal diantaranya:

  1. Proses memang sudah selesai mengerjakan tugasnya, sehingga diakhiri secara normal
  2. Melewati batas waktu yang telah diberikan.

3. Terjadi kesalahan perhitungan misalnya mengerjakan instruksi pembagian dengan nol (division by zero) , atau menyimpan angka yang lebih besar daripada yang dapat diakomodasi oleh perangkat keras.
4. Terjadi kegagalan I/O seperti kegagalan pembacaan dan penulisan file.
5. Proses induknya berakhir, pada kasus ini suatu proses dibuat oleh proses lain, proses pembuat disebut sebagai *parent*, sedangkan proses yang dibuat disebut sebagai *child*. Sistem dirancang untuk mengakhiri secara otomatis proses – proses childnya bila proses parent berakhir.
6. Proses child diakhiri atas permintaan proses parentnya. Pada kasus ini parent mengirim signal tertentu untuk mengakhiri childnya misalnya mengirim signal SIGQUIT, SIGKILL atau SIGTERM.

3. Konsep Pembuatan Proses

Konsep pembuatan proses pada sistem operasi linux :

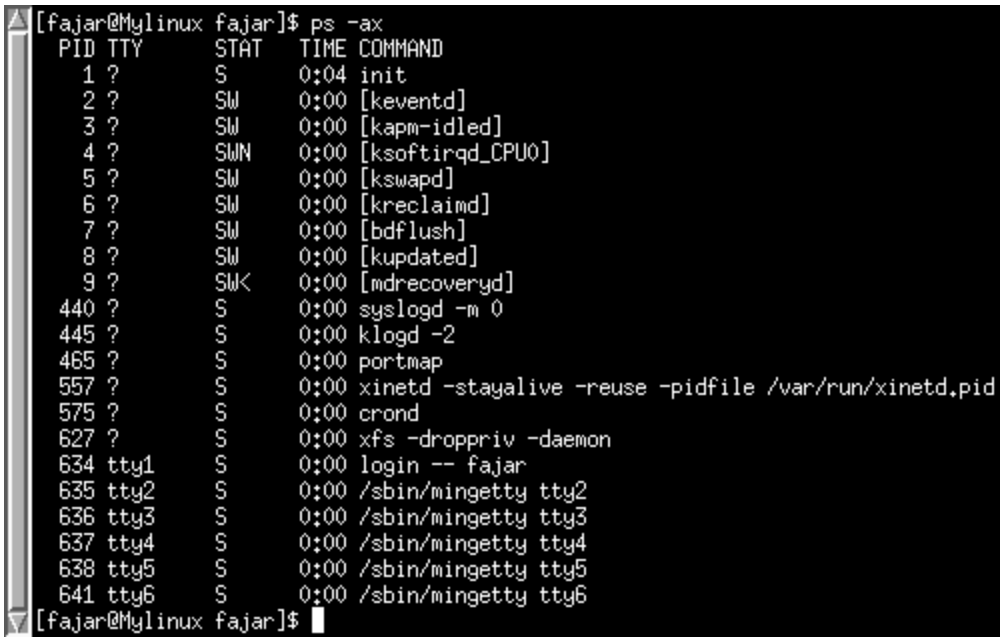
- Setiap proses diberi nomor khusus sebagai identifikasi yang disebut process identification atau **PID** berupa angka integer unik.
- Jika proses selesai (Terminated) maka semua sumber daya yang digunakan termasuk **PID** dibebaskan kembali.
- Proses dibuat menggunakan system call fork() yang sering disebut *forking* proses
- System call fork() mengkopi proses pemanggil sehingga akan terdapat 2 proses yaitu :

  1. Proses pemanggil disebut PARENT
  2. Proses hasil kopian disebut CHILD

- Proses CHILD identik dengan proses PARENT-nya tetapi memiliki **PID** yang berbeda.
- Setelah proses baru (child) berhasil dibuat eksekusi dilanjutkan secara normal di masing – masing proses pada baris setelah pemanggilan system call fork().
- Proses pemanggil (PARENT) dapat melakukan forking proses lebih dari satu kali sehingga memungkinkan terdapat banyak proses CHILD yang dieksekusi.
- Proses CHILD dapat melakukan forking proses seperti halnya PARENT sehingga dapat terbentuk struktur pohon proses.

4. Identifikasi Proses

Untuk melihat **PID** setiap proses di sistem, dapat digunakan perintah ps seperti terlihat pada gambar berikut :



Gambar 1. Output ps

Pustaka GNU C telah menyediakan beberapa tipe dan fungsi untuk membaca ID proses, diantaranya :

- Tipe Data : pid_t

Tipe yang merepresentasikan ID proses, tipe data **pid_t** merupakan signed integer dalam pustaka GNU tipe ini adalah int.

- Fungsi : pid_t getpid (void)

Fungsi '**getpid**' mengembalikan ID proses aktif dan tidak membutuhkan argumen

- Fungsi : pid_t getppid (void)

Fungsi `**getppid**' (get parent pid) mengembalikan ID parent proses aktif dan tidak membutuhkan argumen

Untuk menggunakan tipe data dan fungsi diatas perlu disertakan file header **types.h** dan **unistd.h** Berikut contoh program yang yang membaca ID proses aktif dan ID proses parent-nya.

```
/* contoh1.c */

#include <stdio.h>                      /* standard I/O         */
#include <unistd.h>                     /* getpid(), getppid()  */
#include <sys/types.h>                  /* pid_t                */

int
```

```
main() {

     pid_t mypid, myparentpid;            /*   deklarasi   variabel   penampung
       */

     mypid=getpid();                      /* ambil ID proses ini        */
     myparentpid=getppid();               /* ambil ID parent proses ini */

     /* tampilkan ID proses ini     */
     printf("PID Saya = %d\n",mypid);
     /* tampilkan ID parent         */
     printf("PID PARENT Saya = %d\n",myparentpid);
}
```

Lakukan proses kompilasi dan Link program contoh1.c

```
gcc contoh1.c -o contoh1
```

Apabila proses kompilasi dan link sukses eksekusi program

```
./contoh1
```

```
PID Saya = 1028
PID PARENT Saya = 1027
```

Jika anda mencoba program ini maka output diatas tentunya dapat saja berbeda dengan output di sistem anda. Pemanggilan fungsi pada bahasa C dapat disederhanakan dengan cara berikut:

```
printf("PID Saya = %d\n",getpid());   /* tampilkan PID proses ini */
printf("PID PARENT Saya = %d\n",getppid());/* tampilkan PID parent  */
```

5. Membuat Proses

Untuk pembuatan proses baru di sistem, GNU C menyediakan system call **fork()** yang disertakan pada pustaka **unistd.h** dengan bentuk prototype sebagai berikut :

```
pid_t fork (void);
```

fork() mengembalikan suatu besaran bertipe **pid_t** yang dapat bernilai :

-1 menandakan pembuatan proses baru (child) gagal
0 menandakan proses child
selain itu merupakan proses parent dan jika variabel PID diakses akan berisi PID child.
Apabila operasi sukses akan terdapat 2 proses yang dieksekusi yaitu proses parent dan proses child, eksekusi akan dilanjutkan pada kedua proses setelah **fork()** sampai selesai. Seperti yang telah dijelaskan bahwa proses parent dan child dapat membuat proses - proses baru yang lain sehingga terdapat banyak proses yang berjalan pada waktu bersamaan (Multitasking). Berikut contoh program yang melakukan forking sebuah proses tanpa memeriksa nilai yang dikembalikan.

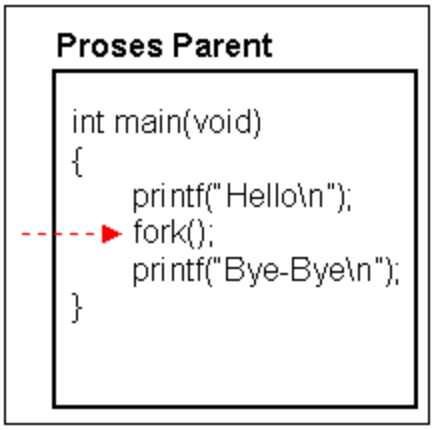
```
/* contoh2.c */

#include <stdio.h>                  /* standard I/O         */
#include <unistd.h>                 /* fork()               */
#include <sys/types.h>              /* pid_t                */

int
main() {

     printf("Hello\n");             /* tampilkan Hello      */
     fork();                        /* buat proses child    */
     printf("Bye-Bye\n");           /* dieksekusi oleh parent dan child */
}
```

```
gcc contoh2.c -o contoh2
```

```
./contoh2
```

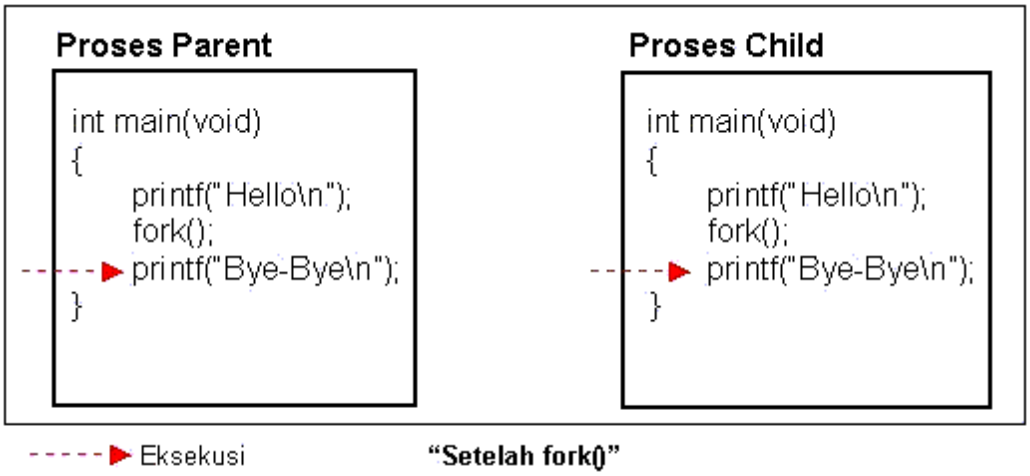
```
Hello
Bye-Bye
Bye-Bye
```

String "Hello" ditampilkan oleh parent sedangkan string "Bye-Bye" masing – masing dieksekusi oleh parent dan child, lebih jelasnya berikut ilustrasi forking proses pada program contoh2.c diatas.



Gambar 2. Proses parent

Setelah menampilkan string "Hello" kelayar, eksekusi berikutnya adalah statement **fork()** yang akan membuat proses child, jika statement ini sukses dieksekusi maka akan terdapat 2 proses yang identik dengan PID yang berbeda. Seperti gambar berikut.



Gambar 3. Eksekusi Setelah fork()

Eksekusi proses PARENT dan proses CHILD dilanjutkan di baris sesudah statement **fork()**, yaitu masing – masing parent dan child menampilkan string "Bye-Bye ". Demikian seterusnya sampai seluruh statement pada parent dan child dieksekusi dan kedua proses berakhir.
Berikut contoh program pembuatan proses baru yang melakukan pemeriksaan nilai kembalian dari **fork()**.

```
/* contoh3.c */

#include <stdio.h>                      /* standard I/O               */
#include <stdlib.h>                     /* standard library           */
#include <unistd.h>                     /*fork(), getpid(), getppid() */
#include <sys/types.h>                  /* pid_t                      */

int
main(){

     pid_t pid;

     pid=fork();                        /* buat proses child  */

     if (pid==-1){
       print("Fork gagal\n");
       exit(EXIT_FAILURE);              /* exit jika gagal            */
     }

     if (pid==0){
       /* proses child */
       print("CHILD: Ini proses Child\n");
       print("CHILD: PID saya             = %d\n",getpid());
       print("CHILD: PID parent saya      = %d\n",getppid());
     } else {
       /* proses parent */
       print("PARENT: Ini proses Parent\n");
       print("PARENT: PID saya            = %d\n",getpid());
       print("PARENT: PID parent saya     = %d\n",getppid());
       print("PARENT: PID child saya      = %d\n",pid);
```

```
    }
}
```

Blok if dapat digantikan dengan struktur switch case seperti berikut :

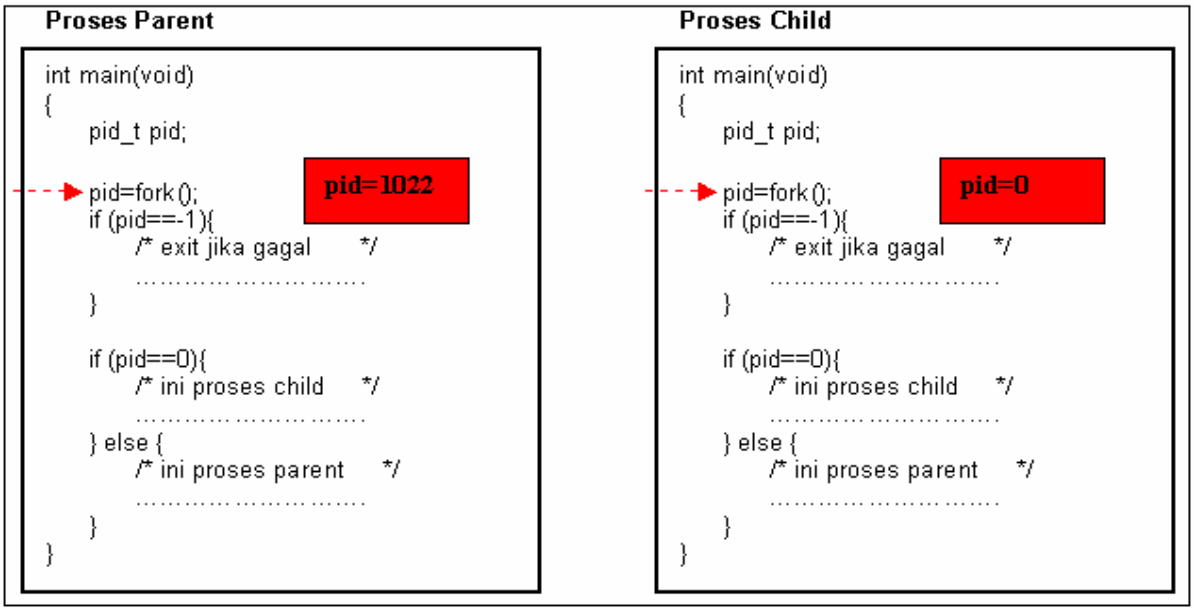
```
    switch(pid) {

            case  -1      : print("Fork gagal\n");
                exit(EXIT_FAILURE);         /* exit jika gagal */

            case  0 : /* proses child */
                print("CHILD: Ini proses Child\n");
                print("CHILD: PID saya            = %d\n",getpid());
                print("CHILD: PID parent saya     = %d\n",getppid());
                exit(EXIT_SUCCESS);              /* child berakhir */

            default : /* proses parent */
                print("PARENT: Ini proses Parent\n");
                print("PARENT: PID saya          = %d\n",getpid());
                print("PARENT: PID parent saya   = %d\n",getppid());
                print("PARENT: PID child saya    = %d\n",pid);
                exit(EXIT_SUCCESS);         /* parent berakhir */

      }
```

Setelah dikompilasi dan link didapat output berikut:

```
./contoh3
```

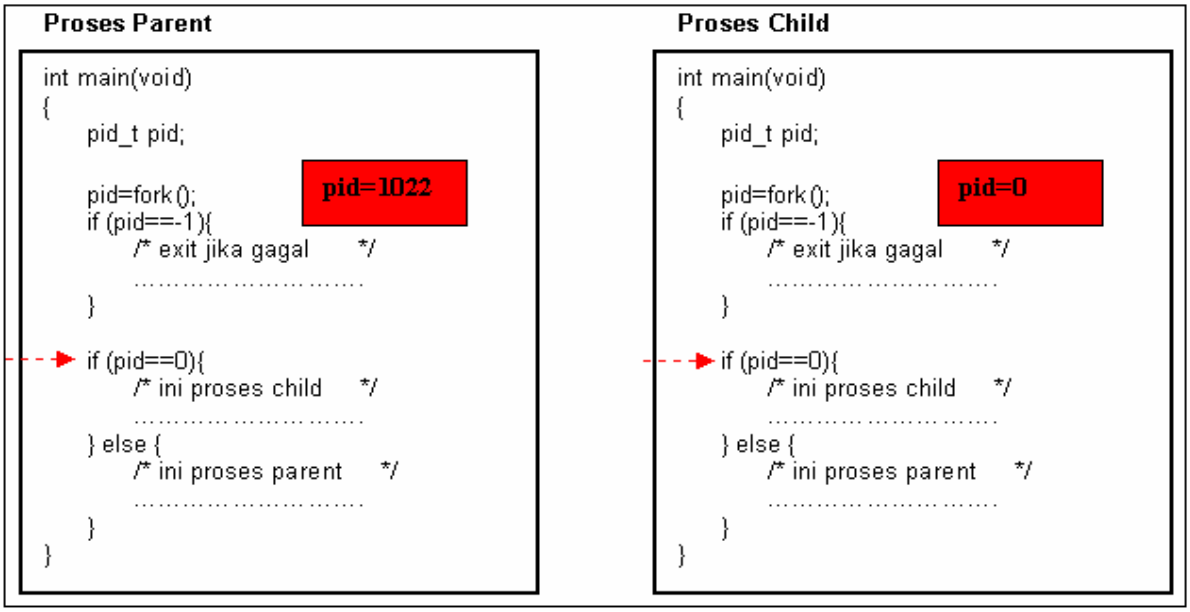
```
PARENT: Ini proses Parent
PARENT: PID saya        = 1021
PARENT: PID parent saya = 1019
PARENT: PID child saya  = 1022
CHILD  : Ini proses Child
CHILD  : PID saya             = 1022
CHILD  : PID parent saya      = 1021
```

Output diatas memperlihatkan forking proses sukses dilakukan, sehingga terdapat 2 proses yang dieksekusi, pada proses parent variabel pid berisi PID child, sedangkan fungsi getppid() pada parent menghasilkan PID parent-nya yaitu PID dari program tempat contoh3 dieksekusi yaitu prompt shell bash. Output diatas dapat saja tidak berurutan dikarenakan quantum time suatu proses telah habis. Sistem operasi linux menggunakan quantum time untuk membatasi setiap proses yang dieksekusi. Ini terkait erat dengan algoritma penjadwalan yang digunakan yaitu Round Robin. Berikut ilustrasi forking proses pada program contoh3.c diatas.



Gambar 4. Proses fork()

Setelah proses forking sukses maka terdapat 2 proses dengan variabel pid-nya masing – masing, variabel pid untuk proses parent berisi pid child pada contoh 1022, sedangkan variabel pid pada proses child berisi nilai 0. eksekusi keduanya dilanjutkan setelah fork yaitu pemeriksaan isi variabel pid, karena proses fork sukses maka variabel pid tentunya tidak bernilai –1 sehingga blok ini tidak akan dieksekusi, tetapi dilanjutkan pada blok if berikutnya. Seperti gambar berikut.



Gambar 5. pemeriksaan variabel pid

Dalam proses parent ekspresi if (pid==0) tidak akan terpenuhi karena variabel pid pada parent bernilai 1022 sehingga statement yang berada dalam blok else yang akan dieksekusi yaitu.

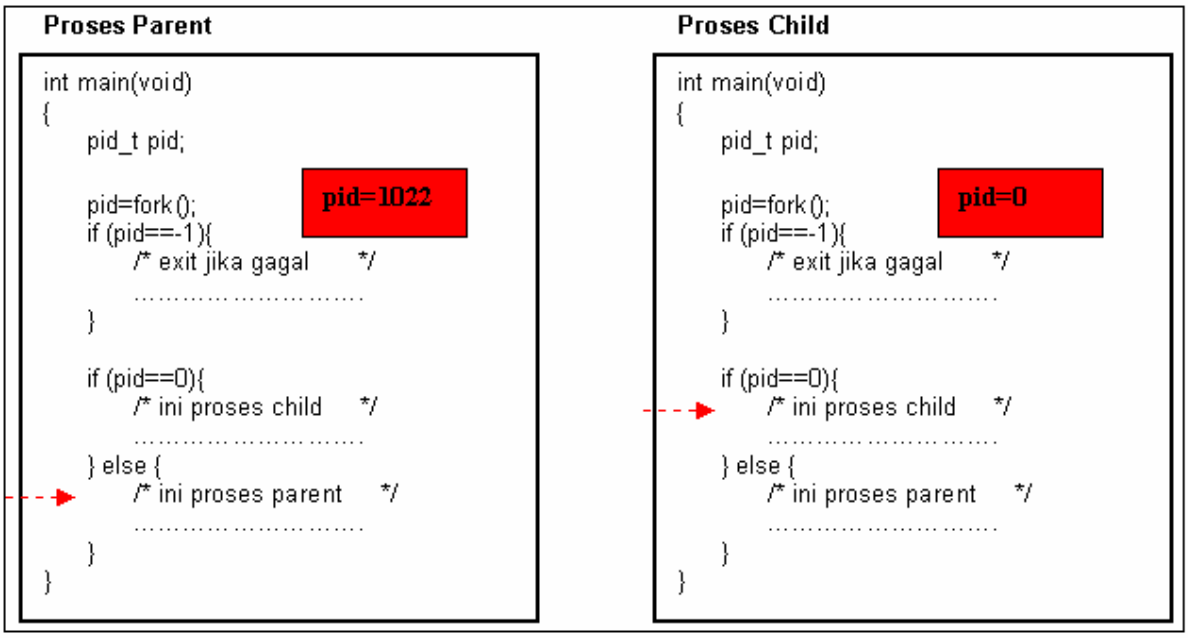
```
print("PARENT: Ini proses Parent\n");
print("PARENT: PID saya              = %d\n",getpid());
print("PARENT: PID parent saya       = %d\n",getppid());
```

```
print("PARENT: PID child saya       = %d\n",pid);
```

Sebaliknya dalam proses child ekspresi if (pid==0) akan terpenuhi dikarenakan variabel pid pada child bernilai 0 sehingga statement yang berada dalam blok akan dieksekusi yaitu

```
print("CHILD: Ini proses Child\n");
print("CHILD: PID saya             = %d\n",getpid());
print("CHILD: PID parent saya      = %d\n",getppid());
```

Berikut ilustrasi eksekusi kedua proses



Gambar 6. Blok yang dieksekusi

Anda dapat menuliskan semua statement yang akan dieksekusi parent maupun child dalam fungsi tertentu seperti contoh berikut.

```
/* contoh4.c */

#include <stdio.h>
```

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

#define MAXLOOP 100

/* prototype fungsi */
void dochild();
void parent();

int
main(){

     pid_t pid;

     pid=fork();

     if(pid==-1){
       perror("Fork gagal:");
       exit(EXIT_FAILURE);
     }

     if(pid==0)
       dochild();  /* eksekusi fungsi dochild    */
     else
       doparent(); /* eksekusi fungsi doparent   */
}


void
dochild(){

      int i;
      for(i=MAXLOOP; i>=1; --i)
            printf("CHILD : %d\n",i);
}

void
doparent(){

      int i;
      for(i=1; i<=MAXLOOP; ++-i)
            printf("PARENT : %d\n",i);
}
```

Output:

./contoh4

```
PARENT   : 1
PARENT   : 2
...........
...........
PARENT   : 100
CHILD    : 100
CHILD    : 99
```

```
...........
...........
CHILD   : 1
```

Baik parent ataupun child dapat melakukan proses forking lebih dari satu kali sehingga akan terbentuk banyak proses baru yang melakukan tugas – tugas tertentu, berikut contoh sederhana yang melakukan forking sebanyak 2x.

```
/* contoh5.c */

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int
main(){

    printf("Hello\n");
    fork();
    printf("Hiii\n");
    fork();
    printf("Hooo\n");

}
```

Output program:

```
./contoh5
```

```
Hello
Hiii
Hiii
Hooo
Hooo
Hooo
Hooo
```

Contoh berikut memperlihatkan 3 proses yang berjalan bersamaan yang melakukan tugas – tugas tertentu yaitu.

- Proses Parent melakukan pembacaan file dan menampilkannya ke layar
- Proses child1 (parent dari child2) melakukan perekaman string ke file data1.txt
- Proses child2 melakukan perekaman string ke file data2.txt

Anda dapat melihat bagaimana mekanisme multitasking yang cukup sederhana berjalan di sistem operasi linux menggunakan system call fork().

```
/* contoh6.c */

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

/* prototype fungsi */
void doparent();
void dochild1();
void dochild2();

int
main(){

     int rv=0,i;
     pid_t chilpid1,childpid2;

     pid1=fork();                    /* buat proses child1 */
     if(pid1==-1) {
        perror("Fork gagal");
        exit(EXIT_FAILURE);
     }

     if(pid1==0) {
       dochild1();
              pid2=fork();           /* buat proses child2 */
              if(pid2==-1){
          perror("Fork gagal");
              exit(1);
     }

     if(pid2==0)
       dochild2();
     else
            doparent();
}

void
doparent(){

      FILE pf;                              /* pointer file */
      char fname[15], buff;
      printf("Input nama file yang dibaca :");
      scanf("%s",fname);                    /* ambil nama file yang isinya ingin
dibaca*/
      pf=fopen(fname,"r");                  /* buka file untuk dibaca */

      if(pf==NULL){
     perror("PARENT: Error : \n");
     exit(EXIT_FAILURE);             /* exit jika buka file gagal */
      }

      buff=getc(pf);                        /* baca karakter pertama */
      printf("PARENT: ISI FILE yang dibaca\n");
```

```
     while(buff!=EOF){
     putc(buff,stdout);                    /* cetak karakter  */
     buff=getc(pf);     /* baca karakter  berikutnya sampai ketemu EOF */
     }

     fclose(pf);                           /* tutup file */

}

void
dochild1(){

     int i;

     FILE *pf=fopen("data2.txt","w");
     if(pf==NULL){
      printf("CHILD1: Error\n");
      exit(EXIT_FAILURE);
      }

     for(i=1; i<=5; ++i)
      fprintf(pf,"%d.Ini dari child1\n",i);

      fclose(pf);
}

void
dochild2(){

     int i;

     FILE *pf=fopen("data3.txt","w");
     if(pf==NULL){
      printf("CHILD2: Error \n");
      exit(1);
     }

     for(i=5; i>=1; --i)
      fprintf(pf,"%d.Ini dari child2\n",i);


     fclose(pf);

}
```

Output program:

```
./contoh6

Input nama file yang dibaca : data3.txt

PARENT: ISI FILE yang dibaca
5.Ini dari child2
```

```
4.Ini dari child2
3.Ini dari child2
2.Ini dari child2
1.Ini dari child2
```

Setelah fork() yang pertama sukses dilakukan maka proses baru yang telah dibuat yaitu child1 melakukan fork() untuk membuat proses baru yang lain (child2). Selanjutnya ketiga proses yaitu parent, child1 dan child2 akan melakukan tugasnya masing – masing.

6. Wait() Membuat Proses parent menunggu

Apabila anda menginginkan proses parent menunggu sampai proses child selesai maka gunakan fungsi wait() yang tersedia pada file header wait.h, berikut contoh program yang memperlihatkan parent menunggu proses child sampai selesai.

```
/* contoh7.c */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>                                   /* wait()
      */

int
main(){
         pid_t pid;
         int rv;

         pid=fork();

         switch(pid) {
            case -1:
                  perror("fork");               /* pesan kesalahan      */
                  exit(EXIT_FAILURE);           /* exit jika gagal      */
            case 0:
                  printf(" CHILD: Ini proses child!\n");
            sleep(5);                           /* tunggu 5 second      */
            printf(" CHILD: Selesai...!\n");
            exit(EXIT_SUCCESS);
            default:
                  printf("PARENT: Ini proses parent!\n");
                  printf("PARENT:     Sekarang     menunggu     child     untuk
selesai...\n");
                  wait(&rv);        /* tunggu sampai child selesai */
                  printf("PARENT: Selesai...!\n");
         }
}
```

Output:

```
./contoh7
```

```
PARENT: Ini proses parent!
PARENT: Sekarang menunggu child untuk selesai...
CHILD: Ini proses child!
CHILD: Selesai...!
PARENT: Selesai...!
```

Terlihat jelas bahwa setelah fungsi wait() dieksekusi oleh proses parent maka parent akan menunggu proses child selesai yaitu mengeksekusi fungsi exit(). Setelah child selesai maka proses parent melanjutkan eksekusinya pula sampai selesai. Pada pustaka GNU C disediakan beberapa makro untuk membaca nilai exit proses child salah satunya adalah makro WEXITSTATUS() yang ada pada wait.h. Berikut contoh program yang membuat proses parent menunggu sampai child selesai dan membaca nilai exit proses child.

```
/* contoh8.c */

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

int
main(){

pid_t pid;
int rv;

 switch(pid=fork()) {
      case -1:
           perror("fork");                  /* pesan kesalahan */
           exit(EXIT_FAILURE);         /* exit jika gagal */
      case 0:
           printf(" CHILD: Ini proses child!\n");
           printf(" CHILD: PID saya adalah %d\n", getpid());
           printf(" CHILD: PID parent saya %d\n", getppid());
           printf(" CHILD: Masukkan status exit saya: ");
           scanf(" %d", &rv);
           printf(" CHILD: Selesai...!\n");
      exit(rv);
      default:
   printf("PARENT: Ini proses parent!\n");
           printf("PARENT: PID saya adalah %d\n", getpid());
           printf("PARENT: PID child saya %d\n", pid);
           printf("PARENT:     Sekarang     menunggu     child     untuk
      selesai...\n");
           wait(&rv); /* tunggu proses child selesai */
           printf("PARENT:status      exit      child      saya      :
      %d\n",WEXITSTATUS(rv));  /*status exit*/
           printf("PARENT: Selesai...!\n");
```

```
  }
}
```

Output:

./contoh8

```
PARENT: Ini proses parent!
PARENT: PID saya adalah 1454
PARENT: PID child saya 1455
PARENT: Sekarang menunggu child untuk selesai...
 CHILD: Ini proses child!
 CHILD: PID saya adalah 1455
 CHILD: PID parent saya 1454
 CHILD: Masukkan status exit saya: 4
 CHILD: Selesai...!
PARENT: status exit child saya : 4
PARENT: Selesai...!
```

Sebelum proses child selesai maka pemakai diminta untuk memasukkan nilai exit yang dapat bernilai dari angka 0 s/d 255. Umumnya nilai exit ini digunakan sebagai informasi penyebab proses child berakhir.

7. Inter Process Communication (IPC)

Proses – proses yang berjalan di sistem dapat berupa proses yang independent atau proses yang saling bekerjasama. Jika 2 atau lebih proses saling bekerjasama berarti keduanya saling berkomunikasi, terdapat beberapa mekanisme komunikasi yang dapat dilakukan seperti komunikasi menggunakan socket dijaringan dengan protokol tertentu, komunikasi melalui mekanisme pipe atau komunikasi melalui pengiriman signal. Tulisan ini akan memperkenalkan kepada anda bagaimana komunikasi antar proses secara internal (IPC) menggunakan signal. Bagi pemakai sistem operasi linux ataupun unix tentunya sudah sering menggunakan perintah sistem kill untuk mengirim signal tertentu kesebuah proses, biasanya perintah kill ini digunakan untuk menghentikan eksekusi suatu proses (termination). Seperti contoh berikut:

Gambar 7. Pengiriman signal dengan kill

Perintah kill membutuhkan argumen berupa nomor PID proses, pada contoh , proses dengan PID 773, secara default jika signal tidak dispesifikasikan maka digunakan SIGTERM dengan nomor 15, hal ini membuat proses tersebut dihentikan. Jadi anda dapat saja menuliskan dengan cara berikut.

```
kill –s SIGTERM 773
```

atau

```
kill –s 15 773
```

Setelah dieksekusi, proses dengan PID 773 akan dihentikan atau diakhiri. Tentunya tidak semua proses yang dapat dihentikan karena akan terkait dengan permission akses yang dimiliki. Jika anda login sebagai root maka anda dapat menghentikan semua proses. Untuk melihat signal yang dapat digunakan oleh perintah kill sertakan opsi –l seperti contoh berikut :

```
kill –l
```

```
1) SIGHUP    2) SIGINT   3) SIGQUIT  4) SIGILL
5) SIGTRAP   6) SIGABRT  7) SIGBUS   8) SIGFPE
 9) SIGKILL 10) SIGUSR1 11) SIGSEGV 12) SIGUSR2
13) SIGPIPE 14) SIGALRM 15) SIGTERM 17) SIGCHLD
……..
54) SIGRTMAX-9    55) SIGRTMAX-8    56) SIGRTMAX-7    57) SIGRTMAX-6
58) SIGRTMAX-5    59) SIGRTMAX-4    60) SIGRTMAX-3    61) SIGRTMAX-2
62) SIGRTMAX-1    63) SIGRTMAX
```

Andapun dapat melihat linux programmer’s manual signal ini via perintah man di prompt shell seperti contoh berikut:

```
man 7 signal
```

```
SIGNAL(7)          Linux Programmer's Manual          SIGNAL(7)

NAME
       signal - list of available signals

DESCRIPTION
       Linux  supports  the  signals listed below. Several signal
       numbers are  architecture  dependent.  First  the  signals
       described in POSIX.1.

       Signal     Value     Action   Comment
       ----------------------------------------------------------------
       SIGHUP      1        A      Hangup detected on controlling terminal
                                   or death of controlling process
       SIGINT      2        A      Interrupt from keyboard
```

```
SIGQUIT     3        C      Quit from keyboard
…    ……    …      …      …      …      …      …      …      …      …      …
SIGKILL     9       AEF     Kill signal
SIGTERM     15        A     Termination signal
SIGCHLD   20,17,18     B    Child stopped or terminated
```

Pustaka GNU C menyediakan beberapa system call untuk pengelolaan proses yang berjalan di sistem. Misalnya fungsi kill() untuk mengirim signal ke suatu proses yang serupa dengan perintah kill. Dengan bentuk prototype sbb:

```
kill(PID,int SIGNAL);
```

dimana :

PID	adalah PID dari proses yang akan dikirimi signal
SIGNAL	adalah signal yang ingin dikirimkan baik berupa nama konstanta ataupun nilainya (lihat jenis – jenis signal pada manual anda)

Berikut contoh program yang serupa dengan perintah kill –s SIGTERM 773

```
/* contoh9.c */

#include <signal.h>                 /* kill()   */
#include <sys/types.h>              /* pid_t    */

int
main(){

        pid_t PID=773;
        kill(PID,SIGTERM);
}
```

Jika dijalankan maka program mencoba mengirim signal termination (SIGTERM) ke proses yang memiliki PID 773. Anda dapat mengubahnya dengan signal SIGQUIT atau SIGKILL yang juga berfungsi untuk menghentikan suatu proses.

Untuk mendefinisikan suatu aksi terhadap signal tertentu anda dapat menggunakan system call sigaction() atau signal(). Sebagai contoh apabila proses child berakhir maka secara otomatis sistem akan mengirim sebuah signal SIGCHLD ke proses parentnya yang menandakan bahwa proses child berakhir, kita dapat mendefinisikan aksi tertentu apabila parent menerima signal ini. Berikut contoh programnya

```
/* contoh10.c */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>                             /* signal()       */
#include <sys/types.h>
#include <sys/wait.h>                           /* wait()              */

/* prototype fungsi penanganan signal  */
void sig_handler();
```

```
int
main(){

     pid_t pid;
     int i;

     signal(SIGCHLD, sig_handler);  /*  install  signal  handler  baru  untuk
SIGCHLD */
     pid = fork();                  /* buat proses baru */

     switch (pid) {
      case  -1: perror("fork");     /* proses fork() gagal  */
                exit(EXIT_FAILURE); /* exit jika gagal      */

       case  0:  /* ini proses child */
                 printf("CHILD: Hello parent\n");
                 sleep(3);           /* tunda selama 3 detik */
                 exit(EXIT_SUCCESS); /* akhiri proses child  */
       default : /* ini proses parent */
                 printf("PARENT: Hello child\n");
                 for (i=1; i<=5; i++) {
                   printf("PARENT: %d\n", i);
                   sleep(1);               /* tunda selama 1 detik */
                 }

                 printf("parent Selesai.\n");
                 exit(EXIT_SUCCESS); /* akhiri proses parent */
     }
}


/* ini definisi fungsi penanganan signal  */
void
sig_handler(){
             printf("child Selesai.\n");
}
```

Output

./contoh10

```
PARENT: Hello child
CHILD  : Hello parent
PARENT: 1
PARENT: 2
PARENT: 3
child selesai.
PARENT: 4
PARENT: 5
```

```
parent selesai.
```

Agar lebih memahami program diatas maka perhatikan alur eksekusinya berikut:

1. Menginstal fungsi penanganan signal SIGCHLD pada parent, hal ini memberitahukan bahwa apabila sistem mengirim SIGCHLD ke parent maka parent mengarahkan aksi penanganannya ke fungsi sig_handler().
2. fork() untuk membuat proses baru (child)
3. Child dibuat tertunda selama 3 detik dan parent melakukan pencetakan nilai i.
4. Setelah 3 detik child mengeksekusi exit() untuk mengakhiri prosesnya
5. Pada saat child selesai sistem mengirim SIGCHLD ke parent dan parent mengarahkannya ke fungsi sig_handler yang menampilkan pesan "child selesai"
6. Parent melanjutkan kembali prosesnya sampai selesai.

Pada contoh program diatas signal dikirim secara otomatis oleh sistem begitu child mengeksekusi fungsi exit, tentunya anda dapat mengirim suatu signal ke proses lain menggunakan system call kill() seperti yang telah kita lakukan, berikut contoh program yang mendefenisikan aksi untuk signal tertentu serta mengirim signal tersebut untuk melihat aksi dari rutin yang disiapkan.

```
/* contoh11.c */

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

void sigquit ();        /*prototype fungsi penanganan signal  */

int
main (){
     pid_t pid;

     pid=fork();

     switch (pid) {
       case -1: perror("fork");            /* proses fork() gagal  */
                exit(EXIT_FAILURE);        /* exit jika gagal      */

       case 0:  /* ini proses child */
                signal (SIGQUIT, sigquit); /* install signal handler baru untuk
child */
                for (;;);                        /* loop terus */

       default: /* ini proses parent */
                printf ("\nPARENT: Kirim signal SIGQUIT ke CHILD\n\n");
                kill (pid, SIGQUIT);       /* kirim signal */
                sleep (3);                 /* tunda 3 detik */
                exit(EXIT_SUCCESS);        /* akhiri proses parent */
     }
}
```

```
/* ini definisi fungsi penanganan signal  */
void
sigquit (){
      printf ("CHILD: Terima SIGQUIT  dari parent.\n");
      exit (0);
}
```

Output

./contoh11

```
PARENT: Kirim signal SIGQUIT ke CHILD
CHILD  : Terima SIGQUIT  dari parent.
```

Tentunya anda dapat mendefinisikan banyak fungsi penanganan signal untuk kebutuhan program seperti contoh berikut ini.

```
/* contoh12.c */

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


/*prototype fungsi penanganan signal  */
void sighup();
void sigint();
void sigquit();
void sigterm();

int
main(){

     pid_t pid;

     pid=fork();

     if ((pid==-1) {
       perror("fork");
       exit(EXIT_FAILURE);
     }


     if (pid == 0) {
       /* install signal habdler*/
                  signal(SIGHUP,sighup);
                  signal(SIGINT,sigint);
                  signal(SIGQUIT, sigquit);
                  signal(SIGTERM, sigterm);
```

```
                for(;;);                /* loop terus menerus   */

    } else {
      /* kirim signal ke child */
      printf("\nPARENT  : kirim signal SIGHUP\n\n");
                kill(pid,SIGHUP);
                sleep(3);                       /* tunda 3 detik        */
                printf("\nPARENT  : kirim signal SIGINT\n\n");
                kill(pid,SIGINT);
                sleep(3);
                printf("\nPARENT  : kirim signal SIGQUIT\n\n");
                kill(pid,SIGQUIT);
                sleep(3);
                printf("\nPARENT  : kirim signal SIGTERM\n\n");
                kill(pid,SIGTERM);
                sleep(3);
      }
}

void
sighup() {
      signal(SIGHUP,sighup); /* reset signal */
      printf("CHILD     : Terima signal SIGHUP\n");
}

void
sigint() {
      signal(SIGINT,sigint);  /* reset signal */
      printf("CHILD     : Terima signal SIGINT\n");
}

void
sigquit(){
      printf("CHILD     : Terima signal SIGQUIT\n");
}

void
sigterm(){
      printf("CHILD     : Terima signal SIGTERM\n");
      exit(EXIT_FAILURE);           /* hentika proses child */
}
```

Output

./contoh12

```
PARENT      : kirim signal SIGHUP
CHILD       : Terima signal SIGHUP
PARENT      : kirim signal SIGINT
CHILD       : Terima signal SIGINT
PARENT      : kirim signal SIGQUIT
CHILD       : Terima signal SIGQUIT
PARENT      : kirim signal SIGTERM
```

```
CHILD       : Terima signal SIGTERM
```

Untuk mengembalikan signal handler ke rutin defaultnya gunakan SIG_DFL , seperti berikut

```
void
sigquit(){
     printf("CHILD   : Terima signal SIGQUIT\n");
     signal(SIGQUIT,SIG_DFL);   /* reset signal ke rutin default */
}
```

Setelah dikembalikan ke rutin defaultnya rutin sigquit tidak akan dipanggil lagi tetapi rutin handler default yang akan dieksekusi, dalam hal ini proses child dihentikan.

Daftar Pustaka

[1] Free Software Foundation.`The GNU C Library Reference Manual', for Version 2.2.x of the GNU C Library. 2001

[2] Stallings, William.‘Operating Systems’, New jersey. Prentice-Hall Inc.1995

[3] Sams Development Team.‘Unix Unleashed’,Sams Publishing.1994

[4] Husain, Kamran. Parker, Tim.‘Red Hat Linux Unleashed’,Sams Publishing.1996

[5] Hariyanto, Bambang.‘Sistem Operasi’, Edisi 2, Informatika Bandung. 2000

Biografi

Mohammad Fajar. Lahir di Toli-toli, Sulawesi tengah 25 November 1977. Lulus SMA Negeri 4 di Makassar tahun 1996, kemudian melanjutkan kuliah di Sekolah Tinggi Manajemen Informatika & Komputer-STMIK Dipanegara Makassar jurusan Teknik Informatika. Lulus S1 tahun 2000 selanjutnya bekerja di PT. INterNUX hingga tahun 2001 dan sekarang menjadi tenaga pengajar di beberapa perguruan tinggi swasta dan pusat pendidikan komputer, diantaranya STMIK Kharisma Belajar pengetahuan komputer secara umum sejak kuliah, ikut aktif dalam kelompok Linux User Group Makassar sebagai ketua litbang periode 2001-2002, dan semenjak kuliah hingga sekarang turut serta dalam beberapa project pembuatan aplikasi. Saat ini mempunyai minat pada bidang software engineering secara umum khususnya system programming. Dapat dihubungi dialamat mail fajarmks@yahoo.com.