ASR4

TD4: Processus sous Unix

Les caractéristiques d'un processus

Écrire un source C générant un exécutable affichant :
- le numéro d'identification du processus
- le numéro d'identification de son père
- le numéro d'identification du propriétaire réel
- le numéro d'identification du propriétaire effectif
- le répertoire de travail
- les temps CPU (utilisateur et noyau) consommés

L'héritage des processus

Écrire un source C qui se dédouble par l'appel à fork. Vérifier que les descripteurs de fichiers sont communs (par exemple, en ouvrant un fichier avant le fork et en lisant dans le fichier après le fork, on se rend compte que la position courante est partagée)
Dans le code du père, faire précéder tous les messages d'information par "Père :".

Dans le code du fils, faire précéder tous les messages d'information par "Fils :".
Faire afficher les informations de l'exercice 1 pour chacun des deux processus.

Synchronisation père/fils

Écrire un source C qui se dédouble et pour lequel le père attend la terminaison du fils pour reprendre son exécution. Faire afficher les informations de l'exercice 1 pour chacun des deux processus.

Recouvrement

Écrire un source C qui se dédouble et pour lequel le fils se recouvre par un nouveau processus. Faire afficher les informations de l'exercice 1 pour chacun des deux processus.

Analyse

Qu'affiche chacun des segments de programme suivants (ce n'est pas forcément la peine de les compiler pour répondre à la question) :

```
(a)
for (i=1; i<=4; i++)
{
  pid = fork();
  if (pid != 0) printf("%d\n", pid);
}

(b)
for (i=1; i<=4; i++)
{
  pid = fork();
  if (pid == 0) break;
  else printf("%d\n", pid);
}

(c)
for (i=0; i<=nb; i++)
{
  p = fork();
  if (p < 0) exit(1);
  execlp("prog", "prog",NULL);
}
wait(&status);

(d)
for (i=1; i<=nb; i++)
{
  p1 = fork();
  p2 = fork();
  if (p1 < 0) exit(1);
  if (p2 < 0) exit(1);
  execlp("prog1", "prog1",NULL);
  execlp("progc", "prog",NULL);
}
wait(&status);
```

(e) Quel est le nombre de processus créés, dans chaque cas ?

AIDE

NAME

times - get process times

SYNOPSIS

```
#include <sys/times.h>

clock_t times(struct tms *buf);
```

DESCRIPTION

times stores the current process times in *buf*.

struct tms is as defined in */usr/include/sys/times.h*:

```
struct  tms  {
```

```
                clock_t tms_utime;  /* user time */
                clock_t tms_stime;  /* system time */
                clock_t tms_cutime; /* user time of children */
                clock_t tms_cstime; /* system time of children */
                };
```

times returns the number of clock ticks that have elapsed since the system has been up.

CONFORMING TO

SVr4, SVID, POSIX, X/OPEN, BSD 4.3

SEE ALSO

time(1), getrusage(2), wait(2)

NAME

getrlimit, getrusage, setrlimit - get/set resource limits and usage

SYNOPSIS

```
#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>

int getrlimit (int resource, struct rlimit *rlim);
int getrusage (int who, struct rusage *usage);
int setrlimit (int resource, const struct rlimit *rlim);
```

DESCRIPTION

getrlimit and **setrlimit** get and set resource limits respectively. *resource* should be one of:

```
RLIMIT_CPU     /* CPU time in seconds */
RLIMIT_FSIZE   /* Maximum filesize */
RLIMIT_DATA    /* max data size */
RLIMIT_STACK   /* max stack size */
RLIMIT_CORE    /* max core file size */
RLIMIT_RSS     /* max resident set size */
RLIMIT_NPROC   /* max number of processes */
RLIMIT_NOFILE  /* max number of open files */
RLIMIT_MEMLOCK /* max locked-in-memory address space*/
```

A resource may unlimited if you set the limit to **RLIM_INFINITY**. **RLIMIT_OFILE** is the BSD name for **RLIMIT_NOFILE**.

The **rlimit** structure is defined as follows :

```
struct rlimit
{
     int  rlim_cur;
     int  rlim_max;
};
```

getrusage returns the current resource usages, for a *who* of either **RUSAGE_SELF** or **RUSAGE_CHILDREN**.

```
struct rusage
{
     struct timeval ru_utime; /* user time used */
     struct timeval ru_stime; /* system time used */
     long ru_maxrss;          /* maximum resident set size */
     long ru_ixrss;      /* integral shared memory size */
     long ru_idrss;      /* integral unshared data size */
     long ru_isrss;      /* integral unshared stack size */
     long ru_minflt;          /* page reclaims */
     long ru_majflt;          /* page faults */
     long ru_nswap;      /* swaps */
     long ru_inblock;         /* block input operations */
     long ru_oublock;         /* block output operations */
     long ru_msgsnd;          /* messages sent */
     long ru_msgrcv;          /* messages received */
     long ru_nsignals;        /* signals received */
     long ru_nvcsw;      /* voluntary context switches */
     long ru_nivcsw;          /* involuntary context switches */
};
```

RETURN VALUE

On success, zero is returned. On error, -1 is returned, and *errno* is set appropriately.

ERRORS

EFAULT *rlim* or *usage* points outside the accessible address space.

EINVAL **getrlimit** or **setrlimit** is called with a bad *resource*, or **getrusage** is called with a bad *who*.

EPERM A non-superuser tries to use **setrlimit()** to increase the soft or hard limit above the current hard limit, or a superuser tries to increase RLIMIT_NOFILE above the current kernel maximum.

CONFORMING TO

SVr4, BSD 4.3

SEE ALSO

ulimit(2), **quotactl**(2)

NAME

getenv - get an environment variable

SYNOPSIS

```
#include <stdlib.h>

char *getenv(const char *name);
```

DESCRIPTION

The **getenv()** function searches the environment list for a string that matches the string pointed to by *name*. The strings are of the form *name* = *value*.

RETURN VALUE

The **getenv()** function returns a pointer to the value in the environment, or NULL if there is no match.

CONFORMING TO

SVID 3, POSIX, BSD 4.3, ISO 9899

SEE ALSO

putenv(3), setenv(3), unsetenv(3), environ(5)

NAME

getcwd, get_current_dir_name, getwd - Get current working directory

SYNOPSIS

```
#include <unistd.h>

char *getcwd(char *buf, size_t size);
char *get_current_working_dir_name(void);
char *getwd(char *buf);
```

DESCRIPTION

The **getcwd()** function copies the absolute pathname of the current working directory to the array pointed to by *buf*, which is of length *size*.

If the current absolute path name would require a buffer longer than *size* elements, **NULL** is returned, and *errno* is set to **ERANGE**; an application should check for this error, and allocate a larger buffer if necessary.

As an extension to the POSIX.1 standard, **getcwd()** allocates the buffer dynamically using **malloc()** if *buf* is **NULL** on call. In this case, the allocated buffer has the length *size* unless *size* is less than zero, when *buf* is allocated as big as necessary. It is possible (and, indeed, advisable) to **free()** the buffers if they have been obtained this way.

get_current_dir_name, which is only prototyped if **__USE_GNU** is defined, will malloc(3) an array big enough to hold the current directory name. If the environment variable **PWD** is set, and its value is correct, then that value will be returned.

getwd, which is only prototyped if **__USE_BSD** is defined, will not malloc(3) any memory. The *buf* argument should be a pointer to an array at least **PATH_MAX** bytes long. **getwd** does only return the first **PATH_MAX** bytes of the actual pathname.

RETURN VALUE

NULL on failure (for example, if the current directory is not readable), with *errno* set accordingly, and *buf* on success.

CONFORMING TO

POSIX.1

SEE ALSO

chdir(2), free(3), malloc(3).