L'ambiente di un processo (I)

- L'ambiente di un processo è un insieme di stringhe (terminate da `\0`).
- Un ambiente è rappresentato da un vettore di puntatori a caratteri terminato da un puntatore nullo.
- Ogni puntatore (che non sia quello nullo) punta ad una stringa della forma:

  identificatore=valore

- Per accedere all'ambiente da un programma C, è sufficiente aggiungere il parametro `envp` a quelli del main:

```
/* showmyenv */
#include <stdio.h>

main(int argc, char **argv, char **envp)
{
    while(*envp)
        printf("%s\n",*envp++);
}
```

  oppure usare la variabile globale seguente:

```
extern char **environ;
```

L'ambiente di un processo (II)

- L'ambiente di default di un processo coincide con quello del processo chiamante.

- Per specificare un nuovo ambiente è necessario usare una delle due varianti seguenti della famiglia `exec`:

  - `execle(path, arg0, arg1, ..., argn, (char *)0, envp);`

  - `execve(path, argv, envp);`

  memorizzando in `envp` l'ambiente desiderato.

L'ambiente di un processo (III)

```
/* setmyenv */
#include <unistd.h>
#include <stdio.h>

main()
{ char *argv[2], *envp[3];

  argv[0] = "setmyenv";
  argv[1] = (char *)0;

  envp[0] = "var1=valore1";
  envp[1] = "var2=valore2";
  envp[2] = (char *)0;

  execve("./showmyenv", argv, envp);
  perror("execve fallita");
}
```

Eseguendo il programma precedente si ottiene quanto segue:

```
>./setmyenv
var1=valore1
var2=valore2
>
```

L'ambiente di un processo (IV)

- Esiste una funzione della libreria standard che consente di cercare all'interno di `environ` il valore corrispondente ad una specifica variabile d'ambiente:

```
#include <stdlib.h>
char *getenv(const char *name);
```

`getenv` prende come argomento il nome della variabile da cercare e restituisce il puntatore al valore (ciò che sta a destra del simbolo =) o `NULL` se non lo trova:

```
#include <stdio.h>
#include <stdlib.h>
main()
{ printf("PATH=%s\n", getenv("PATH"));
}
```

- Dualmente, `putenv` consente di modificare o estendere l'ambiente:
  `putenv("variabile=valore");`

Current working directory e root directory

- Ad ogni processo è associata una **current working directory** che è ereditata dal processo padre. La chiamata di sistema seguente consente di cambiarla:

```
#include <unistd.h>

int chdir(const char *path);
```

- Ad ogni processo inoltre è associata una **root directory** che specifica il punto di inizio (/) del file system visibile dal processo stesso. Per cambiare la root directory si può usare la chiamata di sistema seguente:

```
#include <unistd.h>

int chroot(const char *path);
```

User-id e group-id (I)

- Ad ogni processo sono associati un **real user-id** ed un **real group-id** che coincidono con quelli dell'utente che ha lanciato il processo.

- Esistono anche l'**effective user-id** e l'**effective group-id** che determinano i privilegi del processo. Nella maggior parte dei casi essi coincidono, rispettivamente, con il real user-id ed il real group-id.

- Tuttavia se il file di un programma ha attivo il bit dei permessi noto come set-user-id (set-group-id), allora, quando sarà invocato con una chiamata `exec`, l'effective user-id (effective group-id) diventerà quello del proprietario del file e non quello dell'utente che lo ha lanciato.

User-id e group-id (II)

```
#include <unistd.h>

main()
{
  uid_t uid, euid;
  gid_t gid, egid;

  uid = getuid();
  euid = geteuid();

  gid = getgid();
  egid = getegid();

  uid_t newuid;
  gid_t newgid;
  int status;
  ...
  status = setuid(newuid); /* imposta l'effective user-id */
  status = setgid(newgid); /* imposta l'effective group-id */
}
```

System call per l'accesso a file

Nome	Significato
`open`	apre un file in lettura e/o scrittura o crea un nuovo file
`creat`	crea un file nuovo
`close`	chiude un file precedentemente aperto
`read`	legge da un file
`write`	scrive su un file
`lseek`	sposta il puntatore di lettura/scrittura ad un byte specificato
`unlink`	rimuove un file
`remove`	rimuove un file
`fcntl`	controlla gli attributi associati ad un file

La system call `lseek`

La system call `lseek` permette l'**accesso random** ad un file, cambiando il numero del prossimo byte da leggere/scrivere.

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int filedes, off_t offset, int start_flag);
```

Il parametro `filedes` è un descrittore di file.
Il parametro `offset` determina la nuova posizione del puntatore in lettura/scrittura.
Il parametro `start_flag` specifica da dove deve essere calcolato l'`offset`. `startflag` può assumere uno dei seguenti valori simbolici:

`SEEK_SET` (0)	:	`offset` è misurato dall'inizio del file
`SEEK_CUR` (1)	:	`offset` è misurato dalla posizione corrente del puntatore
`SEEK_END` (2)	:	`offset` è misurato dalla fine del file

`lseek` ritorna la nuova posizione del puntatore.

Offset validi e non validi

Il parametro `offset` può essere **negativo**, cioè sono ammessi **spostamenti all'indietro** a partire dalla posizione indicata da `start_flag`, purchè però non si vada oltre l'inizio del file.
Tentativi di spostamento prima dell'inizio del file generano un errore.

È possibile spostarsi **oltre la fine del file**.
Ovviamente non ci saranno dati da leggere in tale posizione.
Futuri accessi tramite la `read` ai byte compresi tra la vecchia fine del file e la nuova posizione danno come risultato il carattere ASCII null.

Esempio:

```
off_t newpos;
⋮
newpos = lseek(fd, (off_t)-16, SEEK_END);
```

Esempio: gestione di un hotel

Sia `residents` un file contenente la lista dei residenti di un hotel.

La linea 1 di tale file contiene il nome della persona che occupa la camera 1, la linea 2 contiene il nome della persona che occupa la camera 2, ecc.

Ogni linea è lunga 41 caratteri, i primi 40 contengono il nome dell'occupante, l'ultimo è un `newline`.

La funzione `getoccupier`

```
/* getoccupier -- restituisce il nome dell'occupante la camera passata come
parametro */

#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

#define NAMELENGTH 41

char namebuf[NAMELENGTH];   /* buffer per contenere il nome */
int infile = -1;   /* conterra' il file descriptor di residents;
          l'inizializzazione serve affinche' venga aperto una sola volta */

char *getoccupier(int roomno)
{
  off_t offset;
  ssize_t nread;
  if (infile == -1 && (infile = open("residents", O_RDONLY)) == -1)
  {
     return (NULL);
  }
```

. . . codice di `getoccupier`

```
  offset = (roomno -1) * NAMELENGTH;

/* cerca la linea relativa alla camera e legge il nome dell'occupante */
  if (lseek(infile, offset, SEEK_SET) == -1)
     return (NULL);

  if ( (nread = read(infile, namebuf, NAMELENGTH)) <= 0)
     return (NULL);

/* crea una stringa rimpiazzando il newline con un terminatore null */
  namebuf[nread - 1] = '\0';
  return (namebuf);
}
```

Programma per la stampa dei nomi degli occupanti

```
/* stampa i nomi dei residenti in un albergo di 10 camere */

#define NROOMS   10

main()
{
  int j;
  char *getoccupier(int), *p;

  for (j=1; j <= NROOMS; j++)
  {
     if (p = getoccupier(j))
        printf("Room %2d, %s\n", j, p);
     else
        printf("Error on room %d\n", j);
  }
}
```

Esercizi

- Implementare un meccanismo per decidere se una camera è vuota, modificando eventualmente la funzione `getoccupier` e il file `residents`.
  Scrivere una procedura `findfree` per trovare la prima camera libera.

- Scrivere le procedure
  - `freeroom` per rimuovere un occupante da una camera;
  - `addguest` per assegnare una camera ad un ospite, controllando che questa sia libera.

- Utilizzando le funzioni `getoccupier`, `freeroom`, `addguest`, and `findfree`, scrivere un programma `frontdesk` per gestire il file `residents`.