Presentació

Aquesta PAC planteja un seguit d'activitats amb l'objectiu que l'estudiant es familiaritzi amb la temàtica dels darrers mòduls de l'assignatura.

L'estudiant haurà de realitzar un seguit d'experiments i respondre les preguntes plantejades.

La PAC es pot desenvolupar sobre qualsevol sistema Unix (la UOC us facilita la distribució Ubuntu 14.04).

Cada pregunta indica una possible temporització per poder acabar la PAC abans de la data límit i el pes de la pregunta a l'avaluació final de la PAC.

Competències

Transversals:

- Capacitat per a adaptar-se a les tecnologies i als futurs entorns actualitzant les competències professionals

Específiques:

- Capacitat per a analitzar un problema en el nivell d'abstracció adequat a cada situació i aplicar les habilitats i coneixements adquirits per a abordar-lo i resoldre'l
- Capacitat per a dissenyar i construir aplicacions informàtiques mitjançant tècniques de desenvolupament, integració i reutilització

Enunciat

1. Mòdul 5 [Del 14 al 17 de maig] (10%) Responeu **justificadament** les següents preguntes:

   1.1 A què corresponen els descriptors de fitxers oberts que té un procés, a una llista de control d'accés o a una *capability*?

   Corresponen a *capabilities* perque cada cop que el procés hagi d'accedir a un fitxer haurà de presentar el descriptor de fitxer corresponent. Abans d'autoritzar l'accés, el SO validarà si la *capability* permet l'accés sol·licitat.

   Per exemple, si obrim un fitxer en mode `O_RDONLY`, el descriptor de fitxer (*capability*) que retorna la crida al sistema `open` no permetrà realitzar accesos d'escriptura sobre el fitxer.

   1.2 Què és un punt de muntatge? Adjunteu el resultat de l'execució de l'ordre `mount` sobre un sistema linux abans i després de connectar-

hi un pendrive. Destaqueu la línia que indica que s'ha produit el muntatge.

Unix permet integrar diversos sistemes de fitxers, emmagatzemats a diversos dispositius físics (pen drives, CD's, unitats en xarxa, particions del disc dur,...), a un únic sistema de fitxers.

Un punt de muntatge és un directori del sistema de fitxers a partir del qual podrem accedir al sistema de fitxers emmagatzemat a un altre dispositiu físic.

S'adjunta la diferència entre els resultats d'executar la comanda `mount` abans i després de connectar un dispositiu USB al meu computador.

```
/dev/sdd1 on /media/enricm/6680-6A15 type vfat (rw,nosuid,no
dev,relatime,uid=1000,gid=1000,fmask=0022,dmask=0022,codepag
e=437,iocharset=iso8859-1,shortname=mixed,showexec,utf8,flus
h,errors=remount-ro,uhelper=udisks2)
```

La interpretació d'aquesta línea és que el sistema de fitxers de tipus `vfat` emmagatzemat al dispositiu físic `/dev/sdd1` és visible a partir del directori `/media/enricm/6680-6A15`.

2. Mòdul 6 [Del 18 al 20 de maig] (40% = 15% + 25%)

   2.1 Indiqueu **justificadament** quin serà el resultat d'executar els següents programes (nombre de processos creats, parentiu entre ells, informació mostrada per la sortida estàndard). Podeu assumir que les crides al sistema mai tornaran error.

   Observació: És aconsellable que intenteu resoldre aquest tipus de preguntes sense executar els programes a la màquina. Executeu-los per verificar la correctesa de la vostra resposta.

```c
#include <unistd.h>

int
main (int argc, char *argv[])
{
  int fd[2], pid;
  char c;

  pipe (fd);
  pid = fork ();

  if (pid==0)
    write (fd[1], "*", 1);
  else
    {
      read (fd[0], &c, 1);
      write (1, &c, 1);
      write (1, "\n", 1);
    }
}
```

```c
#include <unistd.h>

int
main (int argc, char *argv[])
{
  int fd[2], pid;
  char c;

  pid = fork ();
  pipe (fd);

  if (pid==0)
    write (fd[1], "*", 1);
  else
    {
      read (fd[0], &c, 1);
      write (1, &c, 1);
      write (1, "\n", 1);
    }
}
```

Al codi de l'esquerra, es crea una pipe i un procés fill que comparteix la pipe amb el pare. El fill escriu un caràcter a la pipe i mor. El pare demana llegir un carácter de la pipe; quan el fill l'hagi escrit, el pare el llegirà i l'escriurà pel canal 1 junt a un salt de línia. A continuació, el procés pare morirà.

Al codi de la dreta es crea un procés fill i a continuació pare i fill creen un a pipe, amb el que tindrem dues pipes independents. El fill escriu un caràcter a la seva pipe i mor. El pare sol·licita llegir un caràcter de la seva pipe però com la pipe és buida i existeix un procés escriptor sobre la pipe -el propi procés pare-, el procés pare es quedarà bloquejat indefinidament perquè ningú escriu res a aquesta pipe.

2.2 Escriviu un programa en llenguatge C que, utilitzant les crides al sistema Unix vistes a l'assignatura, comprimeixi un seguit de fitxers. El programa rebrà per la línia de comandes (vector `argv`) la llista de fitxers a comprimir. Per a cada fitxer, caldrà crear un procés fill i executar la comanda `gzip` redireccionant la seva entrada estàndard al fitxer a comprimir i la sortida estàndard al fitxer `/tmp/pid.x` on *pid* és el l'identificador del procés que executa el programa i x és la posició, dins del vector `argv`, del fitxer a comprimir. Caldrà executar totes les compressions concurrentment i, a mesura que finalitzin, el programa haurà d'escriure un missatge indicant el nom del fitxer que ja està comprimit.

Observacions:

- S'adjunta un exemple de l'execució d'aquest programa (línia de comandes `./2 /bin/bash /bin/ls /bin/ps`). Fixeu-vos que les compressions poden finalitzar en un ordre diferent al de creació. També es mostra com comprovar que la compressió s'ha realitzat correctament (si la comanda `cmp` no mostra cap missatge, els fitxers comparats són idèntics).

```
[enricm@willy pac2]$
[enricm@willy pac2]$  ./pac2 /bin/bash /bin/ls /bin/ps
Process 10914 creates process 10915 to compress file /bin/bash
Process 10914 creates process 10916 to compress file /bin/ls
Process 10914 creates process 10917 to compress file /bin/ps
Process 10917 has finished compressing file /bin/ps
Process 10916 has finished compressing file /bin/ls
Process 10915 has finished compressing file /bin/bash
[enricm@willy pac2]$
[enricm@willy pac2]$  ls -l /tmp/10914*
-rw------- 1 enricm enricm 498621 May  7 13:21 /tmp/10914.1
-rw------- 1 enricm enricm  56803 May  7 13:21 /tmp/10914.2
-rw------- 1 enricm enricm  39509 May  7 13:21 /tmp/10914.3
[enricm@willy pac2]$
[enricm@willy pac2]$  gunzip </tmp/10914.1 > xxx
[enricm@willy pac2]$  cmp xxx /bin/bash
[enricm@willy pac2]$
```

- Per generar els noms dels fitxers de sortida podeu utilitzar la rutina de biblioteca `sprintf` i fer quelcom similar a `sprintf(name, "/tmp/%d.%d", getpid(), x);`

- No hi ha definit un límit en el nombre de fitxers a processar. Utilitzeu memòria dinàmica si necessiteu alguna estructura de dades que depengui d'aquest nombre.
- A l'informe de la PAC heu d'annexar una captura de pantalla que mostri el comportament del programa (amb uns fitxers d'entrada diferents als de l'exemple). El codi font del programa s'entregarà per separat.

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

#define BUFFSIZE 128
char buff[BUFFSIZE];

void
panic (char *str)
{
  snprintf (buff, BUFFSIZE, "[%d] (%d:%s) %s\n", getpid (), errno,
            strerror (errno), str);
  write (2, buff, strlen (buff));
  exit (1);
}

int
main (int argc, char *argv[])
{
  int *pid_vector, i;

  if (argc == 1)
    panic ("No arguments");

  pid_vector = malloc ((argc - 1) * sizeof (int));
  if (pid_vector == NULL)
    panic ("Out of memory");

  /* Child creation */
  for (i = 1; i < argc; i++)
    {
      char outname[80];
      int fdin, fdout;

      switch (pid_vector[i - 1] = fork ())
        {
        case -1:
          panic ("fork");

        case 0:
          fdin = open (argv[i], O_RDONLY);
          if (fdin < 0)
            panic ("open input");
          if (dup2 (fdin, 0) < 0)
            panic ("dup2 input");
          if (close (fdin) < 0)
            panic ("close input");
```

```
            sprintf (outname, "/tmp/%d.%d", getppid (), i);
            fdout = open (outname, O_WRONLY | O_TRUNC | O_CREAT, 0600);
            if (fdout < 0)
              panic ("open output");
            if (dup2 (fdout, 1) < 0)
              panic ("dup2 output");
            if (close (fdout) < 0)
              panic ("close output");

            execlp ("gzip", "gzip", NULL);
            panic ("Exec");

          default:
            snprintf (buff, BUFFSIZE,
                      "Process %d creates process %d to compress file %s\n",
                      getpid (), pid_vector[i - 1], argv[i]);
            if (write (2, buff, strlen (buff)) < 0)
              panic ("write");
          }
      }

    /* Child ending */
    for (i = 1; i < argc; i++)
      {
        int pid, st, j;

        pid = wait (&st);
        if (pid < 0)
          panic ("wait");

        for (j = 1; j < argc; j++)
          {
            if (pid == pid_vector[j - 1])
              {
                snprintf (buff, BUFFSIZE,
                          "Process %d has finished compressing file %s\n", pid,
                          argv[j]);
                if (write (2, buff, strlen (buff)) < 0)
                  panic ("write");
              }
          }

      }
    free (pid_vector);
    exit (0);
}
```

3. Mòdul 7 [Del 21 al 28 de maig] (50% = 10% + 40%)

   3.1 Quina és la diferència entre la crida al sistema `pause()` i la crida al sistema `sigsuspend()`?

La crida `pause` bloqueja el procés fins que es dispositi un signal, sense realitzar cap modifiació a la màscara de signals bloquejats. En canvi, la crida `sigsuspend` atòmicamente canvia la màscara de signals bloquejats i bloqueja el procés; quan es rebi un signal, el procés

serà desbloquejat i la màscara de signals bloquejats serà restaurada al valor previ.

3.2 Una aplicació multithread utilitza una estructura de dades ubicada a memòria compartida. L'aplicació es composa dels següents threads:

- `R` threads lectors (reader): periòdicament llegeixen informació de l'estructura de dades.
- `W` threads escriptors (writer): periòdicament modifiquen informació de l'estructura de dades

3.2.1 A la primera versió de l'aplicació no es permetrà cap tipus de concurrència. Únicament un thread (sigui reader o writer) podrà estar accedint a l'estructura. S'adjunta una primera aproximació al codi més rellevant de l'aplicació, però sense utilitzar primitives de semàfors (conseqüentment, el codi és erroni). Es demana que:

- Indiqueu què podria passar si executéssim aquesta versió del codi.
- Afegiu a aquest codi les primitives de semàfors (`sem_init`, `sem_wait`, `sem_signal`) necessàries per garantir el correcte funcionament de l'aplicació. No oblideu les declaracions i inicialitzacions dels semàfors. Si cal, podeu eliminar/reordenar fragments del codi existent però heu de mantenir el codi que realitza l'accés a l'estructura de dades (`get_info`, `put_info`). A aquest i als següents apartats, heu d'evitar qualsevol tipus d'espera activa.

```
/* Shared variables */
global_structure_t gs;
int busy = 0;
```

```
void reader()
{ query_t query; result_t result;

  while (1) {
    query = ...; /* Query generation */

    while (busy == 1) {};
    busy = 1;
    result = get_info(&gs, query);
    busy  = 0;
    /* Process result */
    ...
 }
}
```

```
void writer()
{ int new_value; query_t query;

  while (1) {
    new_value = ...; /* New value generation */
    query = ...;

    while (busy == 1) {};
    busy = 1;
    put_info(&gs, query, new_value);
    busy = 0;
  }
}
```

El codi proposat és incorrecte perquè els accessos i possibles modificacions a la variable entera `busy` no s'estan fent en exclusió mútua, amb el que varis threads podrien accedir a l'estructura de dades concurrentment. A més, els threads en espera estan fent una espera activa.

La solució passa per utilitzar un semàfor per regular l'accés en exclusió mútua a l'estructura y bloquejar els threads en espera.

```
/* Shared variables */
global_structure_t gs;
sem_t mutex_db;

sem_init(mutex_db, 0, 1); /* Init to 1 */
```

```
void reader()
{ query_t query; result_t result;

  while (1) {
    query = ...; /* Query generation */

    sem_wait(mutex_db);
    result = get_info(&gs, query);
    sem_signal(mutex_db);
    /* Process result */
    ...
  }
}
```

```
void writer()
{ int new_value; query_t query;

  while (1) {
    new_value = ...; /* New value genera
    query = ...;

    sem_wait(mutex_db);
    put_info(&gs, query, new_value);
    sem_signal(mutex_db);
  }
}
```

3.2.2 Donat que els threads reader no modifiquen l'estructura de dades, podríem permetre que varis threads reader l'accedissin concurrentment. Ara bé, per garantir la integritat de l'estructura de dades, cal garantir que cada thread writer hi accedirà en exclusió mútua respecte a la resta de threads de l'aplicació. S'adjunta una primera aproximació al codi més rellevant de l'aplicació, però sense utilitzar primitives de semàfors. Conseqüentment, és errònia.

```
/* Shared variables */
global_structure_t gs;
int writer_in = 0, reader_in = 0;
```

Es demana que afegiu a aquest codi les primitives de semàfors necessàries per garantir el correcte funcionament de l'aplicació, prioritzant els threads reader respecte els writer i maximitzant el paral·lelisme entre els threads reader. No oblideu les declaracions i inicialitzacions dels semàfors. Si cal, podeu eliminar/reordenar fragments del codi existent.

```
void reader()
{ query_t query; result_t result;

  while (1) {
    query = ...; /* Query generation */

    while (writer_in == 1) {};
    reader_in ++;
    result = get_info(&gs, query);
    reader_in --;
    /* Process result */
    ...
 }
}
```

```
void writer()
{ int new_value; query_t query;

  while (1) {
    new_value = ...; /* New value generation */
    query = ...;
    while ((writer_in == 1)||(reader_in > 0)) {};
    writer_in = 1;
    put_info(&gs, query, new_value);
    writer_in = 0;
  }
}
```

El canvi respecte a la versió anterior és que si hi ha algun lector accedint a l'estructura, els nous lectors podran accedir-hi directament (sense demanar el mutex sobre l'estructura). Per comprovar-ho afegim un comptador (`reader_in`) que indiqui quants lectors estan accedint a l'estructura, i un nou semàfor mutex per gestionar correctament aquest comptador. En funció del valor d'aquest comptador, els lectors decidiran si cal demanar el mutex sobre tota l'estructura o si es pot accedir directament.

```
/* Shared variables */
global_structure_t gs;
sem_t mutex_db, mutex_ctrl;
int reader_in = 0;

sem_init(mutex_db, 0, 1); /* Init to 1 */
sem_init(mutex_ctrl, 0, 1); /* Init to 1 */
```

```
void reader()
{ query_t query; result_t result;

  while (1) {
    query = ...; /* Query generation */

    entering();

    result = get_info(&gs, query);

    exiting();

    /* Process result */
    ...
  }
}
```

```
void writer()
{ int new_value; query_t query;

  while (1) {
    new_value = ...; /* New value genera
    query = ...;

    sem_wait(mutex_db);
    put_info(&gs, query, new_value);
    sem_signal(mutex_db);
  }
}
```

```
void entering()
{ sem_wait(mutex_ctrl);
  reader_in ++;
  if (reader_in == 1)
    sem_wait(mutex_db);
  sem_signal(mutex_ctrl);
}
```

```
void exiting()
{ sem_wait(mutex_ctrl);
  reader_in --;
  if (reader_in == 0)
    sem_signal(mutex_db);
  sem_signal(mutex_ctrl);
}
```

3.2.3 Suposeu que, per la natura del problema, és possible particionar l'estructura de dades en quatre parts disjuntes i garantir que cada operació tant de lectura com d'escriptura ha d'accedir únicament a dues d'aquestes parts. D'aquesta forma, permetem més accessos concurrents a l'estructura. El codi base seria el següent:

```
/* Shared variables */
global_structure_t gs;
int writer_in[4] = {0, 0, 0, 0}, reader_in[4] = {0, 0, 0, 0};
```

```
void reader2()
{ query_t query;
  result_t result;
  int part1, part2;

  while (1) {
    query = ...; /* Query generation */
    part1 = ...; part2 = ...; /*0,1,2 or 3*/

    while ((writer_in[part1] == 1) ||
           (writer_in[part2] == 1)) {};
    reader_in[part1] ++;
    reader_in[part2] ++;
    result = get_info(&gs, query);
    reader_in[part1] --;
    reader_in[part2] --;
    /* Process result */
    ...
 }
}
```

```
void writer2()
{ int new_value;
  query_t query;
  int part1, part2;

  while (1) {
    new_value = ...; /* New value generation */
    query = ...;
    part1 = ...; part2 = ...; /*0,1,2 or 3*/

    while ((writer_in[part1] == 1) ||
           (writer_in[part2] == 1) ||
           (reader_in[part1] > 0)  ||
           (reader_in[part2] > 0)) {};
    writer_in[part1] = 1; writer_in[part2] = 1;
    put_info(&gs, query, new_value);
    writer_in[part1] = 0; writer_in[part2] = 0;
  }
}
```

Es demana que afegiu a aquest codi les primitives de semàfors necessàries per garantir el correcte funcionament de l'aplicació, prioritzant els threads reader respecte els writer i maximitzant el paral·lelisme entre els threads reader. No oblideu les declaracions i inicialitzacions dels semàfors. Si cal, podeu eliminar/reordenar fragments del codi existent.

El més senzill seria generalitzar la solució de l'apartat b), però replicant quatre cops les variables de controls utilitzades. Ara bé, com es veurà al proper apartat, aquesta solució presenta un problema.

```
/* Shared variables */
global_structure_t gs;
sem_t mutex_db[4], mutex_ctrl[4]; /* All of them init to 1 */
int reader_in[4] = {0, 0, 0, 0};
```

```
void reader2()
{ query_t query;
  result_t result;
  int part1, part2;

  while (1) {
    query = ...; /* Query generation */
    part1 = ...; part2 = ...;/*0,1,2 or 3*/

    entering(part1); entering(part2);

    result = get_info(&gs, query);

    exiting(part1); exiting(part2);

    /* Process result */
    ...
  }
}
```

```
void writer2()
{ int new_value;
  query_t query;
  int part1, part2;

  while (1) {
    new_value = ...; /* New value genera
    query = ...;
    part1 = ...; part2 = ...; /*0,1,2 or

    sem_wait(mutex_db[part1]);
    sem_wait(mutex_db[part2]);

    put_info(&gs, query, new_value);

    sem_signal(mutex_db[part2]);
    sem_signal(mutex_db[part1]);
  }
}
```

```
void entering(int id) /*Must be 0,1,2,3*/
{ sem_wait(mutex_ctrl[id]);
  reader_in[id] ++;
  if (reader_in[id] == 1)
    sem_wait(mutex_db[id]);
  sem_signal(mutex_ctrl[id]);
}
```

```
void exiting(int id) /*Must be 0,1,2,3*/
{ sem_wait(mutex_ctrl[id]);
  reader_in[id] --;
  if (reader_in[id] == 0)
    sem_signal(mutex_db[id]);
  sem_signal(mutex_ctrl[id]);
}
```

3.2.4 Pot donar-se deadlock a la vostra solució de l'apartat c)? En cas negatiu, com es pot demostrar? En cas positiu, modifiqueu la vostra solució de l'apartat c) per tal d'evitar-lo tot utilitzant primitives convencionals de semàfors.

Podria donar-se *deadlock* en cas que els valors emmagatzemats a les variables `part1` i `part2` (representen les parts de l'estructura que estem accedint) no estiguin ordenats per algun criteri. Per exemple, si a un thread writer poden valdre 2 i 3 respectivament, i a un altre thread writer poden valdre 3 i 2 respectivament, el codi anterior pot provocar *deadlock*.

La solució més senzilla passa per ordenar aquests valors utilitzant algun criteri, i demanar els semàfors seguint aquesta ordenació. D'aquesta forma, trenquem el *circular wait*.

```
/* Sorts two integers */
void sort(int *p1, int *p2)
```

```
            {
                if (*p1 < *p2) {
                  int tmp = *p1;
                  *p1 = *p2;
                  *p2 = tmp;
                }
            }
```

```
void reader2()
{ query_t query;
  result_t result;
  int part1, part2;

  while (1) {
    query = ...; /* Query generation */
    part1 = ...; part2 = ...;/*0,1,2 or 3*/

    sort(&part1, &part2);

    entering(part1); entering(part2);

    result = get_info(&gs, query);

    exiting(part1); exiting(pat2);

    /* Process result */
    ...
  }
}
```

```
void writer2()
{ int new_value;
  query_t query;
  int part1, part2;

  while (1) {
    new_value = ...; /* New value genera
    query = ...;
    part1 = ...; part2 = ...; /*0,1,2 or

    sort(&part1, &part2);

    sem_wait(mutex_db[part1]);
    sem_wait(mutex_db[part2]);

    put_info(&gs, query, new_value);

    sem_signal(mutex_db[part2]);
    sem_signal(mutex_db[part1]);
  }
}
```

Recursos

- Mòduls 5, 6 i 7 de l'assignatura.
- Document "Introducció a la programació de UNIX" (disponible a l'aula) o qualsevol altre manual similar.
- L'aula "Laboratori de Sistemes Operatius" (podeu plantejar els vostres dubtes relatius a l'entorn Unix, programació,...).

Criteris d'avaluació

Es valorarà la justificació de les respostes presentades.

El pes de cada pregunta està indicat a l'enunciat.

Format i data de lliurament

Es lliurarà un fitxer `zip` que tingui per nom el vostre identificador en el campus i que contingui un fitxer `pdf` amb la resposta a les preguntes i, si s'escau, els fitxers addicionals que també volgueu lliurar.

Data límit de lliurament: 24:00 del 28 de maig de 2019.