Presentació

Aquesta pràctica planteja un seguit d'activitats amb l'objectiu que l'estudiant pugui aplicar sobre un sistema Unix alguns dels conceptes introduïts als primers mòduls de l'assignatura.

L'estudiant haurà de realitzar un seguit d'experiments i respondre les preguntes plantejades. També haurà d'escriure un petit programa en llenguatge C.

La pràctica es pot desenvolupar sobre qualsevol sistema Unix (la UOC us facilita la distribució Ubuntu 14.04). S'aconsella que mentre realitzeu els experiments no hi hagi altres usuaris treballant al sistema perquè el resultat d'alguns experiments pot dependre de la càrrega del sistema.

Cada pregunta suggereix una possible temporització per poder acabar la pràctica abans de la data límit i el pes de la pregunta a l'avaluació final de la pràctica. El pes d'aquesta pràctica sobre la nota final de pràctiques és del 40%.

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

Per a realitzar pràctica us facilitem el fitxer `pr1so.zip` amb fitxers font. Descompacteu-lo amb la comanda `unzip pr1so.zip`. Per compilar un fitxer, per exemple `prog.c`, cal executar la comanda `gcc -o prog prog.c`

1. Mòdul 2 [Del 8 al 21 de març] (15%)

   Us facilitem els programes `count1.c`, `count2.c` i el shellscript `launch.sh` (no cal que analitzeu com estan implementats).

   - `count1.c` executa un bucle infinit on cada iteració incrementa un comptador (inicialitzat a 0). Quan `count1` rep la notificació asíncrona que indica que ha de finalitzar, imprimeix el valor del comptador i finalitza. `count1.c` emula un procés de càlcul intensiu.

- `count2.c` executa un bucle infinit on cada iteració incrementa un comptador (inicialitzat a 0) i espera un milisegon (bloquejant-se) abans de tornar a iterar. Quan `count2` rep la notificació asíncrona que indica que ha de finalitzar, imprimeix el valor del comptador i finalitza. `count2.c` emula un procés interactiu.
- `launch.sh` inicia l'execució concurrent de `N` processos que executen el programa `count1` i altres `N` que executen `count2` (`N` és un paràmetre del shellscript). Un cop transcorreguts 3 segons, fa finalitzar tots els processos `count`.

A continuació es mostra un exemple de funcionament. Als 3 segons s'escriuen tants comptadors com processos creats (primer els dels processos `count1` i després els dels processos `count2`).

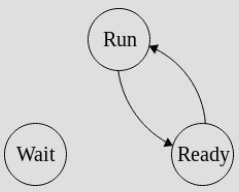
```
[enricm@willy dev]$  ./launch.sh 1
2261259428
3630
[enricm@willy dev]$  ./launch.sh 2
2113053441
2161550926
3787
3787
[enricm@willy dev]$  ./launch.sh 3
1461497516
1465291017
1275001727
3697
3801
3746
[enricm@willy dev]$  ./launch.sh 4
1062104919
1062906744
1036355001
1065605306
3711
3779
3779
3779
[enricm@willy dev]$
```

Compileu `count1.c`, `count2.c` i comproveu que `launch.sh` funciona al vostre sistema de forma similar a l'exemple (haurien d'aparèixer la mateixa quantitat de nombres però els valors concrets seran diferents).

Contesteu les següents preguntes **justificadament**:

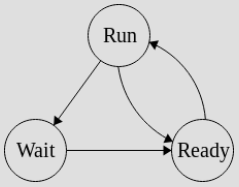
1.1. Estudi dels programes `count`:

1.1.1. A partir de la descripció del comportament de `count1`, dibuixeu un graf amb tres nodes (un per a cada estat possible del procés: Ready, Run i Wait) i amb els arcs que reflecteixin els canvis d'estat que es puguin produir mentre un procés executa `count1`.



En aquest cas el procés mai es bloqueja perquè no sol·licita res al sistema operatiu. Per tant, el procés `count1` només canviarà d'estat des de Run a Ready i a l'inrevés quan el planificador provoqui un canvi de context. El procés morirà quan rebi la notificació asíncrona.

1.1.2. De forma anàloga a 1.1.1., dibuixeu el graf que mostri els canvis d'estat que pot patir un procés que executi `count2`.



En aquest cas el procés demana al SO el servei d'esperar un milisegon amb el que a cada iteració entrarà a l'estat de Wait i l'abandonarà per passar a Ready un cop transcorregut aquest temps. Com al cas anterior, el planificador provocarà canvis d'estat des de Run a Ready i a l'inrevés.

1.2. Anàlisi del hardware:

1.2.1. Indiqueu sobre quin model de processador concret esteu treballant. Per fer-ho, podeu procedir com a aquest exemple:

```
[enricm@willy dev]$  grep "model name" /proc/cpuinfo | head -1
model name      : Intel(R) Core(TM) i5 CPU         650  @ 3.20GHz
[enricm@willy dev]$
```

1.2.2. Quants nuclis físics (*cores*) té el vostre processador? Us pot resultar útil consultar la pàgina `http://ark.intel.com/`.

Consultant la pàgina `http://ark.intel.com/` trobem que Intel i5-650 en té dos. També es pot contestar consultant el fitxer `/proc/cpuinfo`.

1.2.3. Quants nuclis esteu utilitzant realment? Aquest nombre pot ser diferent a l'obtingut a 1.2.2. en funció de com estigui configurat el sistema operatiu, la màquina virtual (en cas que n'estigueu utilitzant una) o si teniu activat el *Hyperthreading*. Per contestar aquesta pregunta compteu quantes línies escriu la següent comanda i adjunteu un screenshot amb el resultat obtingut.

```
grep processor /proc/cpuinfo
```

A la màquina d'exemple s'utilitzen dos nuclis.

1.3. Execució de l'script `launch.sh`

1.3.1. Executeu l'script `launch.sh` diverses vegades, tot variant el valor del paràmetre des de 1 fins al doble del valor que heu contestat a l'apartat 1.2.3.. Expliqueu quina tendència segueixen els comptadors que imprimeixen els processos `count1` (els `N` primers comptadors mostrats) i relacioneu-la amb les respostes 1.2.2. i 1.2.3.. Mostreu screenshots de les execucions.

```
[enricm@willy dev]$  ./launch.sh 1
2253496797
3629
[enricm@willy dev]$  ./launch.sh 2
2168174213
2174924079
3786
3786
[enricm@willy dev]$  ./launch.sh 3
1485044351
1478749401
1400148758
3783
3781
3782
[enricm@willy dev]$  ./launch.sh 4
1069248966
1070725782
1069830904
1090347217
3760
3750
3752
3752
[enricm@willy dev]$  ./launch.sh 5
862627577
848423802
931435667
871977235
850515388
3789
3790
3790
3792
3789
[enricm@willy dev]$
```

Els processos `count1` són de càlcul intensiu, demanen utilitzar la CPU contínuament. Mentre el valor del paràmetre és menor o igual que el nombre de *cores*, cada procés `count1` s'executa en paral·lel sobre un processador físic diferent amb el que els comptadors assoleixen un valor similar (al i5-650, $\approx 2.2 \times 10^9$). Si el valor del paràmetre és superior al nombre de *cores*, tenim més processos `count1` (sol·licitant contínuament ús de processador) que processadors físics, amb el que el planificador de la CPU ha d'executar de forma concurrent els processos sobre els processadors, repartint el temps de processador entre els processos. En aquests casos, si sumem els comptadors de tots els `count1` obtenim un valor similar (al nostre cas, $\approx 4.4 \times 10^9$): la potència global de càlcul del processador ha estat dividida entre varis processos.

1.3.2. Segueixen la mateixa tendència els nombres que imprimeixen els processos `count2` (els `N` darrers comptadors mostrats)? **Justifiqueu** la resposta.

No, el valor dels comptadors `count2` és aproximadament el mateix a tots els casos. Això es degut a que els processos `count2` emulen processos interactius, és a dir, la major part del seu temps de vida estan bloquejats. Per tant, les seves necessitats de CPU són molt inferiors a les dels processos `count1`. Per al nombre de processos `count2` creats, la potència del processador és suficient per a executar-los tots concurrentment.

2. Mòdul 3: Memòria [Del 22 de març al 30 de març] (70% (30% + 40%))

2.1. Us facilitem el programa `stack.c` que admet un paràmetre numèric (1, 2, 3 o 4). Estudieu el seu codi font, compileu-lo i executeu-lo. Us adjuntem uns exemples de la seva execució. És possible que a l'executar-lo al vostre sistema la seqüència de nombres que es genera a cada cas sigui diferent (més llarga o més curta) que la dels exemples; ara bé, en tots els casos el sistema operatiu ha de fer avortar el programa (es mostra el missatge *Segmentation fault*).

```
[enricm@willy dev]$  ./stack 1
Testing rec1...
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 Segmentation
fault (core dumped)
[enricm@willy dev]$  ./stack 2
Testing rec2...
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 Segmentatio
n fault (core dumped)
[enricm@willy dev]$  ./stack 3
Testing rec3...
1 Segmentation fault (core dumped)
[enricm@willy dev]$  ./stack 4
Testing rec4...
0 200 400 600 800 1000 1200 1400 1600 1800 2000 2200 2400 2600 2800 3000 3200 34
00 3600 3800 4000 Segmentation fault (core dumped)
[enricm@willy dev]$
```

Contesteu les següents preguntes:

2.1.1. En tots casos, el programa acaba realitzant un accés a memòria invàlid i el sistema operatiu avorta el procés. Indiqueu **justificadament** en quin punt del programa es provoca aquest accés invàlid en cada cas.

En tots els casos el programa realitza crides recursives, on cada crida recursiva necessita un cert espai de pila per emmagatzemar l'adreça de retorn, variables locals i variables temporals.

Als casos 1, 2 i 3, com la recursivitat no finalitza mai, arriba un moment en que exhaurim l'espai màxim disponible per a la pila i el SO fa avortar el procés. Això es produeix en el moment de fer una crida recursiva perquè en aquest moment s'ha de reservar espai a la pila.

Al cas 4, abans d'omplir la pila es produeix un accés a memòria incorrecte perque intentem escriure més enllà de la posició 99 del vector `global[]` (ubicada a la zona de dades del procés, no en la pila). Quan el SO ho detecta, fa avortar el procés.

2.1.2. A què es degut que el temps que el programa triga en avortar sigui diferent en cada cas? Indiqueu **justificadament** de què depèn?

Als tres primers casos, perquè en cada cas la quantitat d'espai de pila necessari a cada recursió és diferent: `rec1` no declara cap variable local, `rec2` declara un punter (4 o 8 bytes) i demana memòria dinàmica (però aquesta no s'ubica a la pila) i `rec3` declara un vector de 100 integers (400) a cada crida que s'ubica a la pila. Conseqüentment, `rec3` esgotarà la pila abans de `rec2`, i `rec2` abans que `rec1`.

Al cas 4, el SO avorta el procés en el moment que detecta l'accés incorrecte. Degut a que la MMU treballa amb una granularitat de pàgina, típicament l'accés incorrecte no es pot detectar a l'intentar accedir a la posició 100 sinò que es detectarà a l'intentar accedir a la posició 4096.

2.2. Us facilitem l'esquelet del programa `fact.c`. Aquest programa rebrà

com a paràmetre un nombre enter per la línia de comandes (amb valor màxim 20) i escriurà els nombres factorials des de 0 fins a aquest nombre.

2.2.1. Estudieu el codi font facilitat; és un esquelet de programa que haureu de completar. Observeu que l'espai de memòria necessari per emmagatzemar la taula de nombres de la sèrie es vol crear dinàmicament. Observeu també que es pretén emmagatzemar els nombres en dos formats: com a enter de 64 bits i com a string (que també haurà de crear-se dinàmicament en funció del nombre de dígits decimals que tingui el nombre).

2.2.2. Completeu el codi de `fact.c` perquè sol·liciti memòria i calculi la taula de nombres factorials sense malbaratar espai a l'emmagatzemar els factorials com a strings. Un cop omplerta, el codi subministrat la mostrarà per pantalla. Un cop mostrada, el codi alliberarà explícitament tota la memòria que hagi estat demanada. S'adjunta un exemple del resultat desitjat:

```
[enricm@willy dev]$  ./fact 20
0 1 1
1 1 1
2 2 2
3 6 6
4 24 24
5 120 120
6 720 720
7 5040 5040
8 40320 40320
9 362880 362880
10 3628800 3628800
11 39916800 39916800
12 479001600 479001600
13 6227020800 6227020800
14 87178291200 87178291200
15 1307674368000 1307674368000
16 20922789888000 20922789888000
17 355687428096000 355687428096000
18 6402373705728000 6402373705728000
19 121645100408832000 121645100408832000
20 2432902008176640000 2432902008176640000
[enricm@willy dev]$
```

```
/* Your code starts here */
  // Allocate memory
  // Compute fact(n) for i=0 to n
  fact_table =
    (struct fact_entry *) malloc ((n + 1) * sizeof (struct fact_entry));
  if (fact_table == NULL)
    panic ("malloc");

  fact_table[0].n = 0;
  fact_table[0].lli_fact = 1;
  fact_table[0].str_fact = "1";

  for (i = 1; i <= n; i++)
    {
      int digits;

      fact_table[i].n = i;
      fact_table[i].lli_fact = i * fact_table[i - 1].lli_fact;
      digits = log10 (fact_table[i].lli_fact) + 1;
```

```
      fact_table[i].str_fact = malloc ((digits + 1)*sizeof(char));
      if (fact_table[i].str_fact == NULL)
        panic ("malloc");
      sprintf (fact_table[i].str_fact, "%lld", fact_table[i].lli_fact);
    }
/* Your code ends here */
```

```
/* Your code starts here */
  // Free memory
  for (i = 1; i <= n; i++)
    {
      free(fact_table[i].str_fact);
    }
  free(fact_table);
/* Your code ends here */
```

Observacions:

- El vostre codi ha d'estar ubicat entre les línies de `fact.c /* Your code starts here */` i `/* Your code ends here */`.
- No podeu modificar la resta de l'esquelet facilitat i heu de deixar que el codi subministrat mostri el contingut de la taula.
- La rutina `sprintf` pot resultar-vos útil per convertir un nombre des de format enter a format string.
- La rutina `log10` pot resultar-vos útil per calcular el nombre de dígits decimals d'un nombre. Per utilitzar-la, cal afegir `-lm` al compilar, és a dir, `gcc prog.c -lm -o prog`
- Haureu d'entregar el codi font del programa i una captura de pantalla que mostri el seu funcionament.

3. Mòdul 4: Entrada/Sortida [Del 31 de març al al 3 d'abril] (15%)

   El programa `args.c` mostra la llista de paràmetres que rep per la línia de comandes. S'adjunten diversos exemples d'execució:

```
[enricm@willy dev]$  ./args
argc = 1
  argv[0]=./args
[enricm@willy dev]$  ./args a1 a2
argc = 3
  argv[0]=./args
  argv[1]=a1
  argv[2]=a2
[enricm@willy dev]$  ./args a1 a2 | wc
argc = 3
  argv[0]=./args
  argv[1]=a1
  argv[2]=a2
       0       0       0
[enricm@willy dev]$  ./args /bin/l*s
argc = 4
  argv[0]=./args
  argv[1]=/bin/less
  argv[2]=/bin/loadkeys
  argv[3]=/bin/ls
[enricm@willy dev]$
```

   Estudieu el seu codi, compileu-lo i comproveu que funciona con s'indica.

   Contesteu **justificadament** les següents preguntes:

   3.1. Com és que al tercer exemple `argc` té el valor 3 i no 5?

> Es degut a que el símbol `|` és un metacaràcter de l'intèrpret de comandes (shell). En detectar-lo, el shell interpreta el següent paràmetre (`wc`) com a un fitxer executable i comunica mitjançant una pipe la sortida estàndard del procés que executarà `args` amb l'entrada estàndar del procés que executarà `wc`, sense que el procés en sigui conscient (és a dir, no ho veu al vector `argv`).

   3.2. Com és que al quart exemple `argc` té el valor 4 i no 2?

> Es degut a que el símbol `*` és un metacaràcter de l'intèrpret de comandes (shell). En detectar-lo, el shell busca tots els fitxers on el seu nom encaixi amb el patró `/bin/l*s`. Al nostre cas són tres: `/bin/less, /bin/loadkeys, /bin/ls`. Per tant, el shell invoca el programa `./args` passant-li com a paràmetres aquests tres noms de fitxer. Conseqüentment, `argc` valdrà 4.

   3.3. Substituïu a `args.c` les dues aparicions de `stderr` per `stdout`. Compileu el programa i torneu a executar el tercer exemple. Expliqueu a què és degut el nou comportament observat.

> Ara `args` escriu el seu resultat per la sortida estàndard (`stdout`). Com amb el metacaràcter `|` comuniquem la sortida estàndar de `args` amb l'entrada estàndar de `wc`, a aquesta execució tot el que el programa escriu per la seva sortida estàndard va a parar a `wc`. Com `wc` compta el nombre de línies, paraules i caràcters que li arriben per l'entrada estàndar, aquest és el resultat que observem.

```
[enricm@willy dev]$  ./args2 a1 a2 a3 | wc
      5       7      66
[enricm@willy dev]$
```

Recursos

- Mòduls 1, 2, 3 i 4 de l'assignatura.
- L'aula "Laboratori de Sistemes Operatius" (dubtes relatius a Unix, C,...).
- Document "Intèrpret de comandes UNIX" (disponible a l'aula) o qualsevol altre manual similar.
- Qualsevol manual bàsic de llenguatge C.

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 el codi font del programa.

Data límit de lliurament: 24:00 del 3 d'abril de 2019.