CONCEPTES AVANÇATS DE SISTEMES OPERATIUS (CASO)
Facultat d'Informàtica de Barcelona,
Dept. d'Arquitectura de Computadors,
curs 2018/2019 – 2Q
1r Control Parcial 15 de març de 2019

- L'examen és individual.
- Responeu cada pregunta en un full separat.
- Indiqueu a cada full COGNOMS, NOM i DNI.
- L'examen és sense llibres ni apunts, no es poden consultar ordinadors portàtils o mòbils.
- Justifiqueu totes les respostes.
- Temps d'examen: **90 minuts**.

1. (2 punts) Què és un sistema operatiu?

   Un sistema operatiu és un programa que fa d'intermediari entre l'usuari i la màquina. Proporciona un entorn d'execució entre convenient i eficient per executar programes. Gestiona la màquina d'una manera segura i proporciona protecció als usuaris.

2. (2 punts) Enumera i explica tres abstraccions de Mach. Indica una crida al sistema relacionada amb cadascun dels tres conceptes.

   Hi ha 5 cinc conceptes fonamentals o *programming abstractions* a Mach. Aquestes primitives són: **Task**, **Thread**, **Port**, **Message** i **Memory Object**. Expliquem només les tres primeres amb un exemple de *syscall* relacionada.

   1. **Task**: el procés clàssic de Unix, a Mach es divideix en dos. La part que fa de contenidor de recursos, tals com memoria virtual o ports de comunicacions, se'n diu **task**. És una entitat passiva, no s'executa a cap processador.

      ```
      kern_return_t
      task_create (mach_port_t parent_task,
                   boolean_t inherit_memory,
                   mach_port_t* child_task);
      ```

   2. **Thread**: és el segon component del procés. La part activa. L'entorn d'execució d'una task. Cada task pot suportar més d'un **thread** executant-se concurrentment, tots compartint els recursos de la task. Tots els **threads** tenen el mateix espai d'adreces de memòria virtual (VM), però es diferencien en l'estat

d'execució, format per un conjunt de registres, tals com l'stack pointer (SP) i el programm counter (PC).

```
kern_return_t
thread_create (mach_port_t  parent_task,
               mach_port_t* child_thread);
```

3. **Port**: el canal de comunicacions mitjançant el qual es comuniquen dos threads. Un **port** és un recurs, propietat d'una task. Un thread té accès a un **port** pel fet de pertànyer a una task.

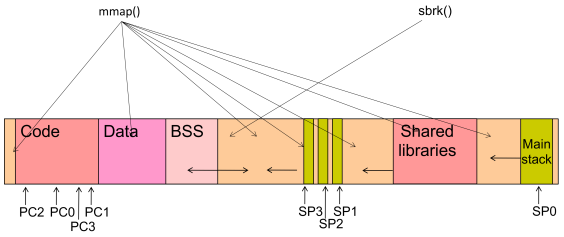
```
kern_return_t
get_privileged_ports (host_priv_t *host_priv_ptr,
                      device_t *device_master_ptr);
```

3. (2 punts) Aquestes dues crides permeten demanar memòria:

```
#include <unistd.h> // change data segment size
void *sbrk(intptr_t increment);
#include <sys/mman.h> // allocate memory, map files or devices into memory
void *mmap(void *addr,size_t length,int prot,int flags,int fd,off_t offset);
```

Fes una comparació de les seves funcionalitats, basada en dibuixos de l'espai d'adreços d'un procés.

La funció sbrk() modifica la mida del segment de dades tot canviant el *program break*, que defineix el final del data segment del procés. Incrementar el *program break* implica alocatar memòria pel procés, mentre que decrementar-lo implica alliberar memòria. Sbrk() retornar l'anterior *program break*.



La funció mmap(), mapeja fitxers o dispositius a memòria tot creant un nou mapatge a la memòria virtual del procés. L'adreça d'inici la pot rebre com a paràmetre, però si aquest és null, el kernel escull la nova adreça. El mapatge es crearà en un límit de pàgina proper. Mmap() retorna l'adreça del nou *mapping*.

4. (2 punts) Explica les diferències que hi ha entre els clones de Linux i els threads de Mach. Quines diferències hi ha entre un thread de Mach i un PThread?

Els **clones** de Linux provenen de la funció clone() que crea un procés nou. Depenent de la parametrització de la funció, el procés nou pot compartir parts del seu context d'execució amb el procés que ha cridat a clone(). Per obtenir una situació similar amb Mach, haurem de crear una **task**, amb task_create() i després crear un **thread** d'aquesta task, amb thread_create(), tots els threads d'una task comparteixen el

mateix context d'execució. Cal notar que, a Linux, la crida fork() s'implementa amb clone(), parametritzada de tal manera que el procés resultant de la crida no comparteix res amb el procés que ha fet la crida.

A Mach, un cop creat el thread cal fixar el seu estat cridant a thread_set_state(), que implica un coneixement profund de l'arquitectura on s'executa. Un cop fixat, cal posar el thread en execució, cridant a thread_resume(). Els threads de la llibreria **Pthread** es creen fent transparent aquesta seqüència d'execució al programador i embolcallant-la en una única crida a pthread_create(). L'objectiu és la portabilitat entre sistemes operatius. En tot cas, pthread_create(), sobre Mach, acabarà cridant a thread_create de sistema. Analogament, pthread_create(), sobre Linux, acabarà cridant a clone().

5. (2 punts) A continuació tens un codi que no és *thread safe*.

```
while (lock==1) ; //spin
lock = 1;
// regio critica de codi
lock = 0;
```

Enumera i explica quins aspectes problemàtics li trobes i proposa un codi alternatiu.

Tres problemes:

1. El compilador pot modificar el codi, buscant optimitzar-lo. Podem aprofitar el suport del compilador amb l'atribut volatile, que indica que un altre flux pot estar accedint a la mateixa variable al mateix temps.
2. L'execució de l'entrada a la regió crítica no és atòmica. Podem fer servir intrínseques del compilador (gcc)

   ```
   while (__sync_lock_test_and_set (&lock, 1)==1);
   ```

3. Sobrecàrrega. Evitar la sobrecàrrega d'instruccions en els multicore d'Intel: instrucció PAUSE. Evitar la sobrecàrrega del bus, per la transacció atòmica: Test, test-and-set.

I una solució:

```
volatile int lock __attribute__ ((aligned(128)));
while (__sync_lock_test_and_set (&sync_var, BUSY)==BUSY)
      while (sync_var==BUSY) asm __volatile__ (\pause");
```