PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ
FACULTAD DE CIENCIAS E INGENIERÍA

SISTEMAS OPERATIVOS
3ra práctica (tipo a)
(Segundo semestre de 2011)

Horario 0781: prof. V. Khlebnikov
Horario 0782: prof. F. Solari A.

Duración: **1 h. 50 min.**
Nota: **No se puede usar ningún material de consulta**
La presentación, la ortografía y la gramática influirán en la calificación.
Puntaje total: **20 puntos**

Pregunta 1 (2 puntos) Responda las siguientes cuestiones:

a) (1 punto) En un sistema que la memoria se organiza con particiones fijas de diferentes tamaños, y los programas son relocalizables, ¿qué problema de desperdicio se presenta? Explique cómo se produce este problema.

b) (1 punto) Los programas con direcciones 'absolutas' no pueden ser ejecutados en ninguna otra parte de la memoria que aquella para la que fueron compilados y enlazados. ¿Cómo ayuda el *hardware* a saltar esta limitación? Explique con un ejemplo breve.

Pregunta 2 (2 puntos) Una memoria organizada por bloques de algún tamaño de asignación menor a los tamaños requeridos, hace que a cada pedido se le entreguen varios bloques contiguos, creándose así las particiones dinámicas y que puede administrarse comúnmente por mapas de bits o por listas.

a) (1.5 puntos) Dado un tamaño de memoria total, el correspondiente mapa de bits requerido ¿es de tamaño fijo o variable? ¿Y la lista? Discuta la conveniencia de uno y otro caso.

b) (0.5 punto) Para una memoria de 64 KiB y bloques de 0.25 KiB, calcule el número de byte y bit en un bitmap, para el bloque en la dirección 0xC040

Pregunta 3 (6 puntos) El siguiente extracto de código de *alloc.c*, parte del *process manager (PM task)* de Minix 2.x y 3.0.x es invocado dentro del PM cuando se libera memoria, por ejemplo, cuando algún proceso termina y el sistema ya puede recuperar el espacio de memoria ocupado. Como se desprende de la descripción mínima de la función, la administración se hace con lista de 'agujeros' o *hole list* cuyos elementos tienen un campo *h_base* , otro campo *h_lenght* y un campo *h_next* que son la dirección inicial, el tamaño y un puntero al siguiente elemento de *hole_list* cuyo primer elemento es apuntado por un puntero global *hole_head.* Otro puntero *free_slots* apunta a elementos nuevos libres para ser usados de ser necesarios.

La función recibe la dirección inicial (medida en *clicks*) y el tamaño (también en *clicks*) del bloque que se está liberando.

```
/*===========================================================================*
 *                          free_mem                                         *
 *===========================================================================*/
PUBLIC void free_mem(base, clicks)
phys_clicks base;            /* base address of block to free */
phys_clicks clicks;          /* number of clicks to free */
{
/* Return a block of free memory to the hole list.  The parameters tell where
 * the block starts in physical memory and how big it is.  The block is added
 * to the hole list.  If it is contiguous with an existing hole on either end,
 * it is merged with the hole or holes.
 */
  register struct hole *hp, *new_ptr, *prev_ptr;

  if (clicks == 0) return;
  if ( (new_ptr = free_slots) == NIL_HOLE)
       panic(__FILE__,"hole table full", NO_NUM);
  new_ptr->h_base = base;
  new_ptr->h_len = clicks;
  free_slots = new_ptr->h_next;
  hp = hole_head;

  /* If this block's address is numerically less than the lowest hole currently
```

```
   * available, or if no holes are currently available, put this hole on the
   * front of the hole list.
   */
  if (hp == NIL_HOLE || base <= hp->h_base) {
        /* Block to be freed goes on front of the hole list. */
        new_ptr->h_next = hp;
        hole_head = new_ptr;
        merge(new_ptr);
        return;
  }

  /* Block to be returned does not go on front of hole list. */
  prev_ptr = NIL_HOLE;
  while (hp != NIL_HOLE && base > hp->h_base) {
        prev_ptr = hp;
        hp = hp->h_next;
  }

  /* We found where it goes.  Insert block after 'prev_ptr'. */
  new_ptr->h_next = prev_ptr->h_next;
  prev_ptr->h_next = new_ptr;
  merge(prev_ptr);           /* sequence is 'prev_ptr', 'new_ptr', 'hp' */
}
```

a) (3 puntos) Explique claramente lo que realiza el código para el propósito buscado, diferenciando y considerando los casos. Ayúdese de algún esquema con los elementos de la lista.

b) (3 puntos) La función *merge()* , que es parte del mismo código fuente en *alloc.c* , ¿cómo es usada en *free_mem()*? ¿Cómo debe operar esta función de manera que cumpla el objetivo en los casos posibles sólo con un argumento?

<u>Pregunta 4</u> (10 puntos) Considere los siguientes datos que presenten el uso de la memoria (con páginas de 4 KiB) durante la ejecución de la orden *cat*:

```
$ cat /proc/self/maps
address           perms  offset dev   inode      pathname
08048000-08051000 r-xp 00000000 08:07 1441834    /bin/cat
08051000-08052000 r--p 00008000 08:07 1441834    /bin/cat
08052000-08053000 rw-p 00009000 08:07 1441834    /bin/cat
086fe000-0871f000 rw-p 00000000 00:00 0          [heap]
b7329000-b7448000 r--p 002a3000 08:07 3804765    /usr/lib/locale/locale-archive
b7448000-b7648000 r--p 00000000 08:07 3804765    /usr/lib/locale/locale-archive
b7648000-b7649000 rw-p 00000000 00:00 0
b7649000-b77a3000 r-xp 00000000 08:07 5374093    /lib/i386-linux-gnu/libc-2.13.so
b77a3000-b77a4000 ---p 0015a000 08:07 5374093    /lib/i386-linux-gnu/libc-2.13.so
b77a4000-b77a6000 r--p 0015a000 08:07 5374093    /lib/i386-linux-gnu/libc-2.13.so
b77a6000-b77a7000 rw-p 0015c000 08:07 5374093    /lib/i386-linux-gnu/libc-2.13.so
b77a7000-b77aa000 rw-p 00000000 00:00 0
b77bc000-b77bd000 r--p 004ed000 08:07 3804765    /usr/lib/locale/locale-archive
b77bd000-b77bf000 rw-p 00000000 00:00 0
b77bf000-b77c0000 r-xp 00000000 00:00 0          [vdso]
b77c0000-b77dc000 r-xp 00000000 08:07 5376119    /lib/i386-linux-gnu/ld-2.13.so
b77dc000-b77dd000 r--p 0001b000 08:07 5376119    /lib/i386-linux-gnu/ld-2.13.so
b77dd000-b77de000 rw-p 0001c000 08:07 5376119    /lib/i386-linux-gnu/ld-2.13.so
bff18000-bff39000 rw-p 00000000 00:00 0          [stack]
```

a) (4 puntos) ¿Cuán grande es el tamaño de memoria que ocupa el programa, en páginas y en KiB?

b) (2 puntos) ¿Qué porcentaje (con resolución hasta las décimas) de la tabla de páginas de un solo nivel ocupan las entradas correspondiente al proceso? Para el cálculo considere los siguientes valores: $2^{10} = 1024$, $2^{20} = 1048576$, $2^{30} = 1073741824$.

c) (4 puntos) Si la tabla de páginas es de 2 niveles, con el tamaño de tablas iguales en cada nivel (4 KiB), ¿qué entradas de estas tablas se usarán para el segmento de texto del programa *cat* (sin incluir *heap*) (2 puntos)? ¿Y para el segmento contiguo más grande (2 puntos)? Para esta última pregunta, tome en cuenta la cantidad de entradas en cada tabla del nivel inferior, y por eso un segmento puede ocupar varias tablas del nivel inferior. Le pueden ser de utilidad los siguientes valores decimales: 0xd7 = 215, 0x3aa = 938.

------- o -------

La preguntas han sido preparadas por VK (4) y FS (1-3).

Profesores del curso: (0781) V. Khlebnikov,
(0782) F. Solari A.

Pando, 26 de octubre de 2011