PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ
FACULTAD DE CIENCIAS E INGENIERÍA

SISTEMAS OPERATIVOS

3ra práctica (tipo a)
(Segundo semestre de 2018)

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 (10 puntos – 50 min.) (*Tanenbaum, A. and Woodhull, A. – Minix 2.0.x – OSDI-2E*) En páginas anexas al final del tema, se tiene el código, numerado por líneas, del archivo `src/mm/alloc.c` correspondiente a los fuentes del sistema operativo Minix 2.0.x. Este archivo tiene las funciones que implementa el manejador de memoria del sistema operativo, como bien indican los comentarios iniciales, para atender las necesidades de espacio de memoria en las llamadas al sistema *fork()* o *exec()* y también en el caso de terminación de procesos como *exit()* o *_exit()*.

Minix 2.0.x se ejecuta en modo protegido de 32 bits, pero sobre memoria real, es decir que las direcciones de memoria corresponden a las direcciones físicas, en un espacio de direccionamiento de 32 bits, con un valor base y un valor límite. Se utiliza una unidad abstracta, *clicks,* para referirse a estas direcciones y como unidad de asignación de memoria. En otros archivos del código fuente, esto se define como 1024, 2048 o similar. El tipo *phys_clicks* corresponde a un entero de 4 bytes que almacena un valor en *clicks*. Los prototipos de las funciones que son punto de entrada, se encuentran en el encabezado *"*`mm.h`*"*.

Para las siguientes cuestiones, considere que un *click* es 1024 bytes, si fuera necesario, y que luego de `mem_init()` y que el sistema ya está corriendo varios procesos, se tienen varios espacios libres no-contiguos en los elementos de `hole[]` usados como una lista, apuntada por `hole_head` y ordenada por las direcciones de memoria, expresadas en *clicks*.

a) (1 punto – 5 min.) Establezca el tamaño en bytes del arreglo de estructura `hole[NR_HOLES]`.

b) (2 puntos – 10 min.) ¿Qué algoritmo de asignación se utiliza en `alloc_mem(clicks)`? Justifique en base al código, y al uso de punteros, no sólo por lo indicado en el comentario. Indique también el valor devuelto y cómo se actualiza el elemento libre restante.

c) (1 punto – 5 min.) ¿Qué sentido tiene usar la función `del_slot(prev_ptr, hp)` en `alloc_mem(clicks)`?

d) (2 puntos – 10 min.) La función `free_mem(base, clicks)` se usa al liberarse algún espacio de memoria. Explique los dos casos que considera para el nuevo elemento, suponiendo que `new_ptr != NIL_HOLE`. Sólo mencione el uso de `merge(new_ptr)` o de `merge(prev_ptr)` que es la siguiente pregunta.

e) (3 puntos – 15 min.) La función `merge(hp)` se utiliza en `free_mem` con dos punteros diferentes. Explique el uso en cada caso, así como el sentido de usar `del_slot(hp, next_ptr)` dentro de esta función.

f) (1 punto – 5 min.) ¿En qué caso podría ser `new_ptr == NIL_HOLE`? ¿Qué significa esto en cuanto al uso de memoria? ¿Qué acción realiza el código y qué significa?

Pregunta 2 (10 puntos – 50 min.) Escribiremos un pequeño programa para investigar las direcciones de memoria que usará este:

```
$ uname -a  # imprimir la información sobre el sistema
Linux kaperna 4.15.0-38-generic #41-Ubuntu SMP Wed Oct 10 10:59:38 UTC 2018 x86_64 x86_64 x86_64 GNU/Linux

$ gcc --version  # imprimir la información sobre la versión del compilador
gcc (Ubuntu 7.3.0-27ubuntu1~18.04) 7.3.0
Copyright (C) 2017 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

```
$ cat -n maxsizes.c | expand
     1  #include <sys/resource.h>
     2  #include <stdint.h>
     3  #include <stdio.h>
     4  #include <stdlib.h>
     5  #include <unistd.h>
     6
     7  #define K 1024
     8  #define M K*K
     9  #define G M*K
    10
    11  int main(void) {
    12      struct rlimit stlim;
    13      char ar_stack[8*M-16*K];
    14      char path_buf[0x100] = {};
    15
    16      printf("SIZE_MAX  = 0x%lx = %ld\n",SIZE_MAX,SIZE_MAX);
    17      printf("int size  = %ld\n",sizeof(int));
    18      printf("long size = %ld\n",sizeof(long));
    19      printf("char size = %ld\n",sizeof(char));
    20
    21      getrlimit(RLIMIT_STACK,&stlim);
    22      printf("Stack limit (soft) = %ld (%ldK, %ldM)\n",
    23             stlim.rlim_cur,stlim.rlim_cur/1024,stlim.rlim_cur/(1024*1024));
    24
    25      printf("stlim    adr = %p\n",&stlim);
    26      printf("ar_stack adr = %p\n",ar_stack);
    27      printf("path_buf adr = %p\n",path_buf);
    28
    29      printf("Page size: %d\n\n",getpagesize());
    30      sprintf(path_buf,"cat /proc/%u/maps",getpid());
    31      system(path_buf); printf("\n");
    32
    33      return 0;
    34  }
```

```
$ gcc maxsizes.c -o maxsizes
$ ./maxsizes
SIZE_MAX  = 0xffffffffffffffff = -1
int size  = 4
long size = 8
char size = 1
Stack limit (soft) = 8388608 (8192K, 8M)
stlim    adr = 0x7ffc418f5c90
ar_stack adr = 0x7ffc418f5da0
path_buf adr = 0x7ffc418f5ca0
Page size: 4096

56303763b000-56303763c000 r-xp 00000000 08:08 13505144                   /home/vk/clases/so/progs/maxsizes
56303783b000-56303783c000 r--p 00000000 08:08 13505144                   /home/vk/clases/so/progs/maxsizes
56303783c000-56303783d000 rw-p 00001000 08:08 13505144                   /home/vk/clases/so/progs/maxsizes
563038e18000-563038e39000 rw-p 00000000 00:00 0                          [heap]
7f568a91c000-7f568ab03000 r-xp 00000000 08:07 3670237                    /lib/x86_64-linux-gnu/libc-2.27.so
7f568ab03000-7f568ad03000 ---p 001e7000 08:07 3670237                    /lib/x86_64-linux-gnu/libc-2.27.so
7f568ad03000-7f568ad07000 r--p 001e7000 08:07 3670237                    /lib/x86_64-linux-gnu/libc-2.27.so
7f568ad07000-7f568ad09000 rw-p 001eb000 08:07 3670237                    /lib/x86_64-linux-gnu/libc-2.27.so
7f568ad09000-7f568ad0d000 rw-p 00000000 00:00 0
7f568ad0d000-7f568ad34000 r-xp 00000000 08:07 3670229                    /lib/x86_64-linux-gnu/ld-2.27.so
7f568af0a000-7f568af0c000 rw-p 00000000 00:00 0
7f568af34000-7f568af35000 r--p 00027000 08:07 3670229                    /lib/x86_64-linux-gnu/ld-2.27.so
7f568af35000-7f568af36000 rw-p 00028000 08:07 3670229                    /lib/x86_64-linux-gnu/ld-2.27.so
7f568af36000-7f568af37000 rw-p 00000000 00:00 0
7ffc418f4000-7ffc420f3000 rw-p 00000000 00:00 0                          [stack]
7ffc42105000-7ffc42108000 r--p 00000000 00:00 0                          [vvar]
7ffc42108000-7ffc4210a000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

Al final de programa se imprimen los mapas de todos los segmentos del programa. Mientras que las primeras dos columnas indican las direcciones del inicio y del fin (excluyendo) de cada segmento, la tercera columna indica los bits de protección de cada segmento que ayudan a identificarlos. Así, el primer segmento es ejecutable y no modificable, este es el segmento de texto, el código (instrucciones) del programa. El 2do es el segmento de datos (no son ejecutables) inicializados (no pueden ser modificados). El 3er segmento es de datos no inicializados (BSS). Después sigue *heap*. Desde el 5to segmento están las librerías dinámicas mapeadas. Y etc.

a) (3 puntos – 15 min.) Con el cálculo a partir de las direcciones del inicio y del fin de los segmentos, indique cuántas páginas ocupa el segmento *heap*, el segmento *stack*, y todos (absolutamente todos) los segmentos con el código ejecutable del programa.

Cada proceso tiene límites de los recursos del sistema. En la línea 21 se consulta el límite de la pila que se asigna a cada proceso. Se puede ver la pila es de 8 MB. Entonces, crear un arreglo grande dentro de la pila no es una buena idea. Mejor sería usar el segmento de datos o el segmento *heap* que puede crecer. Pero, en nuestro caso, usaremos la pila para el arreglo. Si declaramos el arreglo de 8 MB, obtendremos el error *Segmentation fault* durante la ejecución del programa porque otras variables también necesitan un espacio en la pila. Por eso el tamaño del arreglo está establecido en (8MB – 16KB).

b) (2 puntos – 10 min.) Relativo al inicio de la pila, ¿cuáles son (*page*, *offset*) de las variables `stlim`, `ar_stack`, `path_buf`? ¿Cuántas páginas ocupa el arreglo?

La computadora que usamos tiene unos 16 GB de memoria RAM:

```
$ grep MemTotal /proc/meminfo
MemTotal:       16216600 kB
```

Modificamos el programa al siguiente:

```
$ cat -n maxsizes_2.c | expand
     1  #include <stdio.h>
     2  #include <stdlib.h>
     3  #include <unistd.h>
     4
     5  #define K 1024
     6  #define M K*K
     7  #define G M*K
     8  #define S G
     9
    10  char ar_bss[S][32];
    11
    12  int main(void) {
    13      int i,j;
    14      char path_buf[0x100] = {};
    15
    16      printf("ar_bss[0][0] adr = %p\n",&ar_bss[0][0]);
    17      printf("ar_bss[U][U] adr = %p\n",&ar_bss[S-1][31]);
    18      printf("path_buf adr = %p\n",path_buf);
    19
    20      printf("Page size: %d\n\n",getpagesize());
    21      sprintf(path_buf,"cat /proc/%u/maps",getpid());
    22      system(path_buf); printf("\n");
    23
    24      for (i=0; i<S; i++)
    25          for (j=0; j<32; j++)
    26              ar_bss[i][j] = 'a'+(i+j)%26;
    27      printf("ar_bss[U][U] = %c%c%c\n",
    28             ar_bss[S-1][29],ar_bss[S-1][30],ar_bss[S-1][31]);
    29
    30      return 0;
    31  }
```

```
$ gcc maxsizes_2.c -o maxsizes_2
$ time ./maxsizes_2
ar_bss[0][0] adr = 0x55d9f823a040
ar_bss[U][U] adr = 0x55e1f823a03f
path_buf adr = 0x7ffc3ed7be60
Page size: 4096

55d9f8039000-55d9f803a000 r-xp 00000000 08:08 13505146                   /home/vk/clases/so/progs/maxsizes_2
55d9f8239000-55d9f823a000 r--p 00000000 08:08 13505146                   /home/vk/clases/so/progs/maxsizes_2
55d9f823a000-55d9f823b000 rw-p 00001000 08:08 13505146                   /home/vk/clases/so/progs/maxsizes_2
55d9f823b000-55e1f823b000 rw-p 00000000 00:00 0
55e1f84b8000-55e1f84d9000 rw-p 00000000 00:00 0                          [heap]
7f863f9b8000-7f863fb9f000 r-xp 00000000 08:07 3670237                    /lib/x86_64-linux-gnu/libc-2.27.so
7f863fb9f000-7f863fd9f000 ---p 001e7000 08:07 3670237                    /lib/x86_64-linux-gnu/libc-2.27.so
7f863fd9f000-7f863fda3000 r--p 001e7000 08:07 3670237                    /lib/x86_64-linux-gnu/libc-2.27.so
7f863fda3000-7f863fda5000 rw-p 001eb000 08:07 3670237                    /lib/x86_64-linux-gnu/libc-2.27.so
7f863fda5000-7f863fda9000 rw-p 00000000 00:00 0
7f863fda9000-7f863fdd0000 r-xp 00000000 08:07 3670229                    /lib/x86_64-linux-gnu/ld-2.27.so
7f863ffa6000-7f863ffa8000 rw-p 00000000 00:00 0
7f863ffd0000-7f863ffd1000 r--p 00027000 08:07 3670229                    /lib/x86_64-linux-gnu/ld-2.27.so
7f863ffd1000-7f863ffd2000 rw-p 00028000 08:07 3670229                    /lib/x86_64-linux-gnu/ld-2.27.so
7f863ffd2000-7f863ffd3000 rw-p 00000000 00:00 0
7ffc3ed5d000-7ffc3ed7e000 rw-p 00000000 00:00 0                          [stack]
7ffc3edf3000-7ffc3edf6000 r--p 00000000 00:00 0                          [vvar]
7ffc3edf6000-7ffc3edf8000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

```
ar_bss[U][U] = opq

real    37m8,014s
user    1m33,109s
sys     0m26,378s
```

c) (2 puntos – 10 min.) Calcule a partir de las direcciones de los inicios y los fines de los segmentos, ¿cuáles y cuántas páginas ocupa el arreglo `ar_bss`?

d) (2 puntos – 10 min.) ¿Cuántos bytes se quedan disponibles en la última página que usa `ar_bss`?

e) (1 punto – 5 min.) Si el primer programa se ejecuta unos 24 milisegundos por primera vez y 1-2 ms las veces siguientes; el segundo programa, como se puede observar en la salida impresa, tomó más de 37 minutos. ¿Qué operación fue tan lenta? Después de la terminación del programa, ¿se quedaron algunas evidencias de los resultados de su ejecución?

La práctica ha sido preparada por FS(1) y VK(2)
en Linux Mint 19 Tara con LibreOffice Writer

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

Pando, 2 de noviembre de 2018

src/mm/alloc.c

```
 1  /* This file is concerned with allocating and freeing arbitrary-size blocks of
 2   * physical memory on behalf of the FORK and EXEC system calls.  The key data
 3   * structure used is the hole table, which maintains a list of holes in memory.
 4   * It is kept sorted in order of increasing memory address. The addresses
 5   * it contains refer to physical memory, starting at absolute address 0
 6   * (i.e., they are not relative to the start of MM).  During system
 7   * initialization, that part of memory containing the interrupt vectors,
 8   * kernel, and MM are "allocated" to mark them as not available and to
 9   * remove them from the hole list.
10   *
11   * The entry points into this file are:
12   *   alloc_mem:   allocate a given sized chunk of memory
13   *   free_mem:    release a previously allocated chunk of memory
14   *   mem_init:    initialize the tables when MM start up
15   *   max_hole:    returns the largest hole currently available
16   */
17
18  #include "mm.h"
19  #include <minix/com.h>
20  #include <minix/callnr.h>
21  #include <signal.h>
22  #include "mproc.h"
23
24  #define NR_HOLES  (2*NR_PROCS)     /* max # entries in hole table */
25  #define NIL_HOLE (struct hole *) 0
26
27  PRIVATE struct hole {
28    struct hole *h_next;             /* pointer to next entry on the list */
29    phys_clicks h_base;              /* where does the hole begin? */
30    phys_clicks h_len;               /* how big is the hole? */
31  } hole[NR_HOLES];
32
33  PRIVATE struct hole *hole_head;    /* pointer to first hole */
34  PRIVATE struct hole *free_slots;/* ptr to list of unused table slots */
35  #if ENABLE_SWAP
36  PRIVATE int swap_fd = -1;  /* file descriptor of open swap file/device */
37  PRIVATE u32_t swap_offset; /* offset to start of swap area on swap file */
38  PRIVATE phys_clicks swap_base;     /* memory offset chosen as swap base */
39  PRIVATE phys_clicks swap_maxsize;/* maximum amount of swap "memory" possible */
40  PRIVATE struct mproc *in_queue;    /* queue of processes wanting to swap in */
41  PRIVATE struct mproc *outswap = &mproc[LOW_USER];  /* outswap candidate? */
42  #else /* !SWAP */
43  #define swap_base ((phys_clicks) -1)
44  #endif /* !SWAP */
45
46  FORWARD _PROTOTYPE( void del_slot, (struct hole *prev_ptr, struct hole *hp) );
47  FORWARD _PROTOTYPE( void merge, (struct hole *hp)                           );
48  #if ENABLE_SWAP
49  FORWARD _PROTOTYPE( int swap_out, (void)                                    );
50  #else
51  #define swap_out()        (0)
52  #endif
```

```
53
54  /*===========================================================================*
55   *                                alloc_mem                                  *
56   *===========================================================================*/
57  PUBLIC phys_clicks alloc_mem(clicks)
58  phys_clicks clicks;               /* amount of memory requested */
59  {
60  /* Allocate a block of memory from the free list using first fit. The block
61   * consists of a sequence of contiguous bytes, whose length in clicks is
62   * given by 'clicks'.  A pointer to the block is returned.  The block is
63   * always on a click boundary.  This procedure is called when memory is
64   * needed for FORK or EXEC.  Swap other processes out if needed.
65   */
66
67    register struct hole *hp, *prev_ptr;
68    phys_clicks old_base;
69
70    do {
71          hp = hole_head;
72          while (hp != NIL_HOLE && hp->h_base < swap_base) {
73                   if (hp->h_len >= clicks) {
74                            /* We found a hole that is big enough.  Use it. */
75                            old_base = hp->h_base;     /* remember where it started */
76                            hp->h_base += clicks;      /* bite a piece off */
77                            hp->h_len -= clicks;       /* ditto */
78
79                            /* Delete the hole if used up completely. */
80                            if (hp->h_len == 0) del_slot(prev_ptr, hp);
81
82                            /* Return the start address of the acquired block. */
83                            return(old_base);
84                   }
85
86                   prev_ptr = hp;
87                   hp = hp->h_next;
88          }
89    } while (swap_out());            /* try to swap some other process out */
90    return(NO_MEM);
91  }
92
93  /*===========================================================================*
94   *                                free_mem                                   *
95   *===========================================================================*/
96  PUBLIC void free_mem(base, clicks)
97  phys_clicks base;         /* base address of block to free */
98  phys_clicks clicks;               /* number of clicks to free */
99  {
100 /* Return a block of free memory to the hole list.  The parameters tell where
101  * the block starts in physical memory and how big it is.  The block is added
102  * to the hole list.  If it is contiguous with an existing hole on either end,
103  * it is merged with the hole or holes.
104  */
105
106   register struct hole *hp, *new_ptr, *prev_ptr;
107
108   if (clicks == 0) return;
109   if ( (new_ptr = free_slots) == NIL_HOLE) panic("Hole table full", NO_NUM);
110   new_ptr->h_base = base;
111   new_ptr->h_len = clicks;
112   free_slots = new_ptr->h_next;
113   hp = hole_head;
114
115   /* If this block's address is numerically less than the lowest hole currently
116    * available, or if no holes are currently available, put this hole on the
117    * front of the hole list.
118    */
119   if (hp == NIL_HOLE || base <= hp->h_base) {
120          /* Block to be freed goes on front of the hole list. */
121          new_ptr->h_next = hp;
122          hole_head = new_ptr;
123          merge(new_ptr);
124          return;
125   }
126
127   /* Block to be returned does not go on front of hole list. */
128   while (hp != NIL_HOLE && base > hp->h_base) {
129          prev_ptr = hp;
130          hp = hp->h_next;
131   }
132
133   /* We found where it goes.  Insert block after 'prev_ptr'. */
134   new_ptr->h_next = prev_ptr->h_next;
135   prev_ptr->h_next = new_ptr;
136   merge(prev_ptr);                /* sequence is 'prev_ptr', 'new_ptr', 'hp' */
137 }
138
139 /*===========================================================================*
140  *                                del_slot                                   *
141  *===========================================================================*/
142 PRIVATE void del_slot(prev_ptr, hp)
143 register struct hole *prev_ptr;     /* pointer to hole entry just ahead of 'hp' */
144 register struct hole *hp;  /* pointer to hole entry to be removed */
145 {
146 /* Remove an entry from the hole list.  This procedure is called when a
147  * request to allocate memory removes a hole in its entirety, thus reducing
```

```
148    * the numbers of holes in memory, and requiring the elimination of one
149    * entry in the hole list.
150    */
151
152     if (hp == hole_head)
153            hole_head = hp->h_next;
154     else
155            prev_ptr->h_next = hp->h_next;
156
157     hp->h_next = free_slots;
158     free_slots = hp;
159   }
160
161   /*===========================================================================*
162    *                                merge                                      *
163    *===========================================================================*/
164   PRIVATE void merge(hp)
165   register struct hole *hp;  /* ptr to hole to merge with its successors */
166   {
167   /* Check for contiguous holes and merge any found.  Contiguous holes can occur
168    * when a block of memory is freed, and it happens to abut another hole on
169    * either or both ends.  The pointer 'hp' points to the first of a series of
170    * three holes that can potentially all be merged together.
171    */
172
173     register struct hole *next_ptr;
174
175     /* If 'hp' points to the last hole, no merging is possible.  If it does not,
176      * try to absorb its successor into it and free the successor's table entry.
177      */
178     if ( (next_ptr = hp->h_next) == NIL_HOLE) return;
179     if (hp->h_base + hp->h_len == next_ptr->h_base) {
180            hp->h_len += next_ptr->h_len;       /* first one gets second one's mem */
181            del_slot(hp, next_ptr);
182     } else {
183            hp = next_ptr;
184     }
185
186     /* If 'hp' now points to the last hole, return; otherwise, try to absorb its
187      * successor into it.
188      */
189     if ( (next_ptr = hp->h_next) == NIL_HOLE) return;
190     if (hp->h_base + hp->h_len == next_ptr->h_base) {
191            hp->h_len += next_ptr->h_len;
192            del_slot(hp, next_ptr);
193     }
194   }
195
196   /*===========================================================================*
197    *                                mem_init                                   *
198    *===========================================================================*/
199   PUBLIC void mem_init(total, free)
200   phys_clicks *total, *free;         /* memory size summaries */
201   {
202   /* Initialize hole lists.  There are two lists: 'hole_head' points to a linked
203    * list of all the holes (unused memory) in the system; 'free_slots' points to
204    * a linked list of table entries that are not in use.  Initially, the former
205    * list has one entry for each chunk of physical memory, and the second
206    * list links together the remaining table slots.  As memory becomes more
207    * fragmented in the course of time (i.e., the initial big holes break up into
208    * smaller holes), new table slots are needed to represent them.  These slots
209    * are taken from the list headed by 'free_slots'.
210    */
211
212     register struct hole *hp;
213     phys_clicks base;                /* base address of chunk */
214     phys_clicks size;                /* size of chunk */
215     message mess;
216
217     /* Put all holes on the free list. */
218     for (hp = &hole[0]; hp < &hole[NR_HOLES]; hp++) hp->h_next = hp + 1;
219     hole[NR_HOLES-1].h_next = NIL_HOLE;
220     hole_head = NIL_HOLE;
221     free_slots = &hole[0];
222
223     /* Ask the kernel for chunks of physical memory and allocate a hole for
224      * each of them.  The SYS_MEM call responds with the base and size of the
225      * next chunk and the total amount of memory.
226      */
227     *free = 0;
228     for (;;) {
229            mess.m_type = SYS_MEM;
230            if (sendrec(SYSTASK, &mess) != OK) panic("bad SYS_MEM?", NO_NUM);
231            base = mess.m1_i1;
232            size = mess.m1_i2;
233            if (size == 0) break;               /* no more? */
234
235            free_mem(base, size);
236            *total = mess.m1_i3;
237            *free += size;
238   #if ENABLE_SWAP
239            if (swap_base < base + size) swap_base = base+size;
240   #endif
241     }
242
```

```
243   #if ENABLE_SWAP
244     /* The swap area is represented as a hole above and separate of regular
245      * memory.  A hole at the size of the swap file is allocated on "swapon".
246      */
247     swap_base++;                              /* make separate */
248     swap_maxsize = 0 - swap_base;             /* maximum we can possibly use */
249   #endif
250   }
251
252   #if ENABLE_SWAP
253   /*===========================================================================*
254    *                                swap_on                                    *
255    *===========================================================================*/
256   PUBLIC int swap_on(file, offset, size)
257   char *file;                               /* file to swap on */
258   u32_t offset, size;                       /* area on swap file to use */
259   {
260   /* Turn swapping on. */
261
262     if (swap_fd != -1) return(EBUSY); /* already have swap? */
263
264     tell_fs(CHDIR, who, FALSE, 0);    /* be like the caller for open() */
265     if ((swap_fd = open(file, O_RDWR)) < 0) return(-errno);
266     swap_offset = offset;
267     size >>= CLICK_SHIFT;
268     if (size > swap_maxsize) size = swap_maxsize;
269     if (size > 0) free_mem(swap_base, (phys_clicks) size);
270   }
271
272   /*===========================================================================*
273    *                                swap_off                                   *
274    *===========================================================================*/
275   PUBLIC int swap_off()
276   {
277   /* Turn swapping off. */
278     struct mproc *rmp;
279     struct hole *hp, *prev_ptr;
280
281     if (swap_fd == -1) return(OK);    /* can't turn off what isn't on */
282
283     /* Put all swapped out processes on the inswap queue and swap in. */
284     for (rmp = &mproc[LOW_USER]; rmp < &mproc[NR_PROCS]; rmp++) {
285           if (rmp->mp_flags & ONSWAP) swap_inqueue(rmp);
286     }
287     swap_in();
288
289     /* All in memory? */
290     for (rmp = &mproc[LOW_USER]; rmp < &mproc[NR_PROCS]; rmp++) {
291           if (rmp->mp_flags & ONSWAP) return(ENOMEM);
292     }
293
294     /* Yes.  Remove the swap hole and close the swap file descriptor. */
295     for (hp = hole_head; hp != NIL_HOLE; prev_ptr = hp, hp = hp->h_next) {
296           if (hp->h_base >= swap_base) {
297                   del_slot(prev_ptr, hp);
298                   hp = hole_head;
299           }
300     }
301     close(swap_fd);
302     swap_fd = -1;
303     return(OK);
304   }
305
306   /*===========================================================================*
307    *                                swap_inqueue                               *
308    *===========================================================================*/
309   PUBLIC void swap_inqueue(rmp)
310   register struct mproc *rmp;               /* process to add to the queue */
311   {
312   /* Put a swapped out process on the queue of processes to be swapped in.  This
313    * happens when such a process gets a signal, or if a reply message must be
314    * sent, like when a process doing a wait() has a child that exits.
315    */
316     struct mproc **pmp;
317
318     if (rmp->mp_flags & SWAPIN) return;       /* already queued */
319
320
321     for (pmp = &in_queue; *pmp != NULL; pmp = &(*pmp)->mp_swapq) {}
322     *pmp = rmp;
323     rmp->mp_swapq = NULL;
324     rmp->mp_flags |= SWAPIN;
325   }
326
327   /*===========================================================================*
328    *                                swap_in                                    *
329    *===========================================================================*/
330   PUBLIC void swap_in()
331   {
332   /* Try to swap in a process on the inswap queue.  We want to send it a message,
333    * interrupt it, or something.
334    */
335     struct mproc **pmp, *rmp;
336     phys_clicks old_base, new_base, size;
337     off_t off;
```

```
338     int proc_nr;
339
340     pmp = &in_queue;
341     while ((rmp = *pmp) != NULL) {
342             proc_nr = (rmp - mproc);
343             size = rmp->mp_seg[S].mem_vir + rmp->mp_seg[S].mem_len
344                     - rmp->mp_seg[D].mem_vir;
345
346             if (!(rmp->mp_flags & SWAPIN)) {
347                     /* Guess it got killed.  (Queue is cleaned here.) */
348                     *pmp = rmp->mp_swapq;
349                     continue;
350             } else
351             if ((new_base = alloc_mem(size)) == NO_MEM) {
352                     /* No memory for this one, try the next. */
353                     pmp = &rmp->mp_swapq;
354             } else {
355                     /* We've found memory.  Update map and swap in. */
356                     old_base = rmp->mp_seg[D].mem_phys;
357                     rmp->mp_seg[D].mem_phys = new_base;
358                     rmp->mp_seg[S].mem_phys = rmp->mp_seg[D].mem_phys +
359                             (rmp->mp_seg[S].mem_vir - rmp->mp_seg[D].mem_vir);
360                     sys_newmap(proc_nr, rmp->mp_seg);
361                     off = swap_offset + ((off_t) (old_base-swap_base)<<CLICK_SHIFT);
362                     lseek(swap_fd, off, SEEK_SET);
363                     rw_seg(0, swap_fd, proc_nr, D, (phys_bytes)size << CLICK_SHIFT);
364                     free_mem(old_base, size);
365                     rmp->mp_flags &= ~(ONSWAP|SWAPIN);
366                     *pmp = rmp->mp_swapq;
367                     check_pending(rmp);        /* a signal may have waked this one */
368             }
369     }
370 }
371
372 /*===========================================================================*
373  *                                swap_out                                   *
374  *===========================================================================*/
375 PRIVATE int swap_out()
376 {
377 /* Try to find a process that can be swapped out.  Candidates are those blocked
378  * on a system call that MM handles, like wait(), pause() or sigsuspend().
379  */
380   struct mproc *rmp;
381   struct hole *hp, *prev_ptr;
382   phys_clicks old_base, new_base, size;
383   off_t off;
384   int proc_nr;
385
386   rmp = outswap;
387   do {
388           if (++rmp == &mproc[NR_PROCS]) rmp = &mproc[LOW_USER];
389
390           /* A candidate? */
391           if (!(rmp->mp_flags & (PAUSED | WAITING | SIGSUSPENDED))) continue;
392
393           /* Already on swap or otherwise to be avoided? */
394           if (rmp->mp_flags & (TRACED | REPLY | ONSWAP)) continue;
395
396           /* Got one, find a swap hole and swap it out. */
397           proc_nr = (rmp - mproc);
398           size = rmp->mp_seg[S].mem_vir + rmp->mp_seg[S].mem_len
399                   - rmp->mp_seg[D].mem_vir;
400
401           prev_ptr = NIL_HOLE;
402           for (hp = hole_head; hp != NIL_HOLE; prev_ptr = hp, hp = hp->h_next) {
403                   if (hp->h_base >= swap_base && hp->h_len >= size) break;
404           }
405           if (hp == NIL_HOLE) continue;      /* oops, not enough swapspace */
406           new_base = hp->h_base;
407           hp->h_base += size;
408           hp->h_len -= size;
409           if (hp->h_len == 0) del_slot(prev_ptr, hp);
410
411           off = swap_offset + ((off_t) (new_base - swap_base) << CLICK_SHIFT);
412           lseek(swap_fd, off, SEEK_SET);
413           rw_seg(1, swap_fd, proc_nr, D, (phys_bytes)size << CLICK_SHIFT);
414           old_base = rmp->mp_seg[D].mem_phys;
415           rmp->mp_seg[D].mem_phys = new_base;
416           rmp->mp_seg[S].mem_phys = rmp->mp_seg[D].mem_phys +
417                   (rmp->mp_seg[S].mem_vir - rmp->mp_seg[D].mem_vir);
418           sys_newmap(proc_nr, rmp->mp_seg);
419           free_mem(old_base, size);
420           rmp->mp_flags |= ONSWAP;
421
422           outswap = rmp;             /* next time start here */
423           return(TRUE);
424   } while (rmp != outswap);
425
426   return(FALSE);  /* no candidate found */
427 }
428 #endif /* SWAP */
```