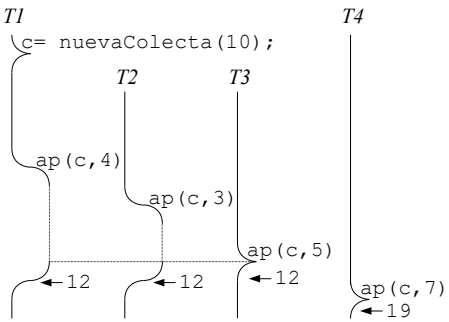
# CC4302 Sistemas Operativos – Control 3 – Semestre Primavera 2025 – Profs.: Mateu, Arenas

## Pregunta 1

Defina el tipo de datos *Colecta* y programe las siguientes funciones:

```
Colecta *nuevaColecta(double meta);
double aportar(Colecta *c, double monto);
```

La función *nuevaColecta* crea y entrega una colecta para juntar al menos *meta* euros. La función *aportar* es invocada por múltiples threads para contribuir con *monto* euros. Esta función debe esperar hasta que la suma de los aportes alcance la meta, entregando el monto total recaudado. Si un thread invoca *aportar* cuando la meta ya fue alcanzada, también se suma a la colecta. La figura de la derecha muestra un ejemplo de ejecución con múltiples threads.




***Restricciones***: Para la sincronización Ud. debe usar spin-locks. El único tipo de busy-waiting permitido es el que hacen los spin-locks. Use una cola *Queue* de spin-locks para los threads en espera.

## Pregunta 2

Programe las funciones *colecta_read* y *colecta_write* para el módulo *colecta*. Al principio viene una primera lectura de 8192 bytes y luego múltiples escrituras. La lectura y las escrituras deben retornar cuando se hayan recolectado 10 bytes o más en escrituras. Una segunda llamada a *colecta_read* debe retornar 0 bytes para señalar el fin de archivo. Ejemplo de uso:

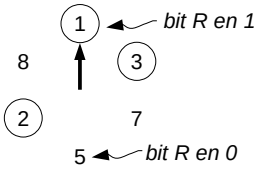
| *Shell 1* | *Shell 2* | *Shell 3* | *Shell 4* |
|---|---|---|---|
| | | | `$ cat < /dev/colecta` |
| `$ echo hola > /dev/colecta` | | | |
| | `$ echo que > /dev/colecta` | | |
| | | `$ echo tal > /dev/colecta` | |
| `$` | `$` | `$` | `hola`<br>`que`<br>`tal`<br>`$` |

Note que el comando *echo* agrega un byte *newline* al final y por lo tanto el 1er. *echo* escribe 5 bytes y el 2do. y 3ero. escriben 4 bytes c/u.

Para la sincronización use los mutex y condiciones de la tarea 7 sobre módulos. Simplifique su solución haciendo que funcione solo con el ejemplo dado. No considere: ejemplos más complejos, manejo de errores, desborde del tamaño del buffer de escritura, condiciones de borde como señales con *control-C*. Los encabezados de las funciones que necesita en su solución están en el reverso de este enunciado.

## Pregunta 3

**I.** Considere un sistema Unix que implementa la estrategia del reloj. El sistema posee 6 páginas reales disponibles y corre un solo proceso. La figura de abajo indica el estado inicial de la memoria, mostrando las páginas residentes en memoria, la posición del cursor y el valor del bit R.

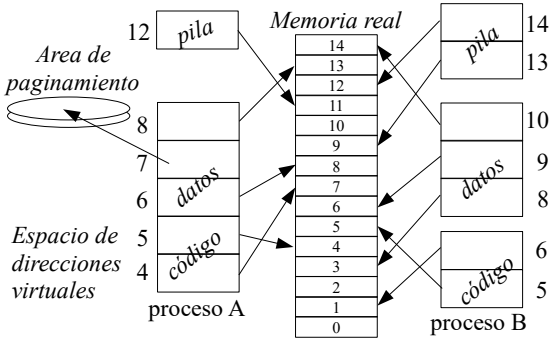



Dibuje los estados por los que pasa la memoria para la siguiente traza de accesos a páginas virtuales: 2, 5, 4, 5, 8, 6.

**II.** Considere un procesador con una MMU que *no implementa* el bit D (*dirty*). Modifique la implementación de la estrategia del reloj que aparece en el reverso de este enunciado de manera que no se grabe una página en disco si esa página ya había sido grabada previamente. No copie toda la implementación, escriba sólo las líneas que modificó más 2 líneas antes y 2 líneas después de la modificación.

*Ayuda*: Use el bit W para emular el bit D y suponga que existe un bit de software W2 que es el que realmente indica si una página se puede escribir o no. Note que el hardware también gatilla un page-fault cuando se escribe en una página cuyo bit V es 1, pero su bit W es 0.

**III.** El diagrama de la derecha muestra la asignación de páginas en un sistema Unix que ejecuta los procesos A y B. Las páginas son de 4 KB. El núcleo utiliza la estrategia *copy-on-write* para implementar *fork*.




(a) Construya la tabla de páginas del proceso A después de que este invoca *sbrk* pidiendo 6 KB adicionales.

(b) A continuación el proceso B invocó *fork*. Construya la tabla de páginas para el proceso hijo justo después de que este modificó la página virtual 9. No construya la tabla del padre. En las tablas indique página virtual, página real y atributos de validez y escritura.

## La estrategia del reloj para un núcleo clásico monocore

```
// Se invoca cuando ocurre un pagefault,
// es decir bit V==0 o el acceso fue una escritura y bit W==0
void pagefault(int page) {
  Process *p= current_process; // propietario de la página
  int *ptab= p→pageTable;
  if (bitS(ptab[page]))        // ¿Está la página en disco?
    pageIn(p, page, findRealPage()); // sí, leerla de disco
  else
    segfault(page);            // no
}
// Graba en disco la página page del proceso q
int pageOut(Process *q, int page) {
  int *qtab= q→pageTable;
  int realPage= getRealPage(qtab[page]);
  savePage(q, page);  // retoma otro proceso
  setBitV(&qtab[page], 0);
  setBitS(&qtab[page], 1);
  return realPage;    // Retorna la página real en donde se ubicaba
}
// Recupera de disco la página page del proceso p colocándola en realPage
void pageIn(Process *p, int page, int realPage) {
  int *ptab= p→pageTable;
  setRealPage(&ptab[page], realPage);
  setBitV(&ptab[page], 1);
  loadPage(p, page);  // retoma otro proceso
  setBitS(&ptab[page], 0);
  purgeTlb();         // invalida la TLB
  purgeL1();          // invalida cache L1
}
Iterator *it; // = processIterator();
Process *cursor_process= NULL;
int cursor_page;
int findRealPage() {
  // recorre las páginas residentes en memoria de todos los procesos
  // buscando una página que no haya sido referenciada
  int realPage= getAvailableRealPage();
  if (realPage≥0)      // ¿Quedan páginas reales disponibles?
    return realPage;   // Sí, retornamos esa página
  // no, hay que hacer un reemplazo
  for (;;) {
    if (cursor_process==NULL) { // ¿Quedan páginas en proceso actual?
      // no, considerar el siguiente proceso
      if (!hasNext(it))     // ¿Quedan procesos por recorrer?
        resetIterator(it); // partiremos con el primer
                           // proceso nuevamente
      cursor_process= nextProcess(it); // pasamos al próximo
      cursor_page= cursor_process→firstPage; // primera pág.
    }
    // Estamos visitando la página cursor_page del proceso cursor_process
    int *qtab= cursor_process→pageTable;
    // mientras queden páginas por revisar en cursor_process
    while (cursor_page≤cursor_process→lastPage) {
      if (bitV(qtab[cursor_page])) {   // ¿Es válida?
        if (bitR(qtab[cursor_page]))   // Sí fue referenciada
          setBitR(&qtab[cursor_page], 0);
        else // no, se reemplaza la pág. cursor_page de cursor_process
          return pageOut(cursor_process, cursor_page++);
      }
      cursor_page++;
    }
    // Se acabaron las páginas de cursor_process,
    // hay que buscar en el próximo proceso
    cursor_process= NULL;
  }
}
```

## Encabezados de funciones para la pregunta 2

```
ssize_t colecta_read( struct file *filp,
            const char *buf, size_t count, loff_t *f_pos);
ssize_t colecta_write( struct file *filp,
            const char *buf, size_t count, loff_t *f_pos);
int colecta_open(struct inode *inode, struct file *filp);
void m_lock(KMutex *mutex);
void m_unlock(KMutex *mutex);
int c_wait(KCondition *cond, KMutex *mutex);
void c_broadcast(KCondition *cond);
void c_signal(KCondition *cond);
size_t copy_to_user(void *to, void *from, ssize_t n);
size_t copy_from_user(void *to, void *from, ssize_t n);
```