Examen de Sistemas Informáticos en Tiempo Real
4º II (18/9/13)

CUESTIONES. TIEMPO: 1 HORA 30 MINUTOS (VALORACIÓN: 50%).

Advertencia: Se piden sobre todo **conceptos**, más que ejemplos concretos de programación, salvo que se pida expresamente.

1. Explique con ayuda de un esquema en qué consiste el direccionamiento virtual y paginado, y qué hay que hacer para que dos **procesos** (no hilos) puedan acceder a variables compartidas.
2. (cambiar)Realice y explique el diseño de un **ejecutivo cíclico** con las actividades que figuran en la tabla. Diga cuáles son los ciclos principal y secundario, así como la distribución temporal de actividades, demostrando que se cumplen las restricciones temporales. Para todas las actividades el tiempo límite ("deadline") es igual a su periodo.

Nombre	Periodo (ms)	Coste (ms)
A	30	7
B	45	5
C	90	10
D	90	20

3. Explique la manera recomendada de esperar una condición mediante variables de condición, y qué sucede desde que el hilo o proceso entra en la espera de dicha variable hasta que sale de ella.
4. Para el siguiente programa explique **en general cómo funciona**, cuánto **tarda** en ejecutarse y qué **mensajes** imprime por pantalla, suponiendo que el proceso que comienza en **main** puede recibir sólo señales de tiempo real entre SIGRTMIN y SIGRTMIN+4. En particular, explique qué sucede cuando **a)** dicho proceso no recibe ninguna señal **b)** recibe las señales SIGRTMIN a los 200 ms y SIGRTMIN+2 a los 400 ms, en ambos casos contando desde el arranque.

```
<cabeceras correctas>
#define N 5
pthread_t h[N]; sigset_t b;
void f1(int a, siginfo_t *b, void *c) {}
void *f(void *p) {
  int k = (int)p;
  struct timespec u = {0, 500000000L};
  if(k>0) pthread_join(h[k-1], NULL);
  printf("k vale %d\n", k);
  nanosleep(&u, NULL);
  return NULL;
}
void *f2(void *p) {
  siginfo_t v; int k;
  while(1) {
    k= sigwaitinfo(&b, &v);
    pthread_cancel(h[k-SIGRTMIN]);
  }
}
int main(int p1, char **p2) {
  int i; struct sigaction c; pthread_t h1;
  sigemptyset(&b);
  c.sa_sigaction = f1; sigemptyset(&c.sa_mask);
  c.sa_flags = SA_SIGINFO;
  for(i=SIGRTMIN; i<=SIGRTMAX; i++) {
    sigaddset(&b, i);
    sigaction(i, &c, NULL);
  }
  sigprocmask(SIG_BLOCK, &b, NULL);
  for(i=0; i<N; i++) {
    pthread_create(&h[i], NULL, f, (void *)i);
  }
  pthread_create(&h1, NULL, f2, NULL);
  pthread_join(h[N-1], NULL);
  return 0;
}
```

5. Para el siguiente programa explique **en general cómo funciona**, cuánto **tarda** en ejecutarse y qué **mensajes** imprime por pantalla, suponiendo el proceso que comienza en **main** sólo recibe señales de tiempo real. En particular, explique qué sucede cuando **a)** se ejecuta el programa como **c2 2 1** y se reciben las señales SIGRTMIN+1, SIGRTMIN+3, SIGRTMIN+2, SIGRTMIN+1 y SIGRTMIN+3 por ese orden, la primera a los 200 ms de arrancar y el resto separadas por 200 ms y **b)** se ejecuta con el mismo comando, pero se reciben las señales SIGRTMIN+1 a los 200 ms y SIGRTMIN+3 a los 400 ms.

```
<cabeceras correctas>
int n = 0;
void f(int a, siginfo_t *b, void *c) {
  n++;
}
int main(int p1, char **p2) {
  int i, a; sigset_t b; struct sigaction c;
  siginfo_t d;
  sigemptyset(&b);
  c.sa_sigaction = f; sigemptyset(&c.sa_mask);
  c.sa_flags = SA_SIGINFO;
  for(i=SIGRTMIN; i<=SIGRTMAX; i++) {
    sigaddset(&b, i); sigaction(i, &c, NULL);
  }
  sigprocmask(SIG_UNBLOCK, &b, NULL);
  for(i=1; i<p1; i++) {
    sscanf(p2[i], "%d", &a);
    sigemptyset(&b);
    sigaddset(&b, SIGRTMIN+a);
    sigprocmask(SIG_BLOCK, &b, NULL);
    while(sigwaitinfo(&b, &d) == -1);
    sigprocmask(SIG_UNBLOCK, &b, NULL);
  }
  printf("n: %d\n", n);
  return 0;
}
```

Datos que pueden ser útiles:

```
int sigaction(int sig, struct sigaction *act, struct sigaction *oact);
pid_t fork(void); int execl(const char *ejecutable, const char *arg0, ...., NULL);
void *malloc(size_t siz); int sscanf(char *origen, char *formato, …);
int kill(pid_t pid, int sig); int sigqueue(pid_t pid, int sig, const union sigval val);
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(* rut_com)(void *), void *arg);
int pthread_join(pthread_t thread, void **valor); int pthread_sigmask(int how, const sigset_t *set, sigset_t *oset);
int pthread_cancel(pthread_t thread); void pthread_testcancel(void);
int sigwaitinfo(const sigset_t *estas_sg, siginfo_t *infop);
int mq_send(mqd_t cola, const char *datos, size_t longitud, unsigned int prioridad);
int mq_getattr(mqd_t cola, struct mq_attr *atributos);
int pthread_setcancelstate(int state, int *oldstate);
state: PTHREAD_CANCEL_ENABLE, PTHREAD_CANCEL_DISABLE
```

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \; ; \; U = \sum_{i=1}^{N} \frac{C_i}{T_i} \; ; \; U_0 = N\left(2^{1/N} - 1\right)$$

```
int sscanf(char  *fuente, char *formato, …);  int sprintf(char *destino, char *formato,…);
int sigemptyset(sigset_t *pset); int sigfillset(sigset_t *pset);
struct sigaction {
  void(* sa_handler) ();
  void (* sa_sigaction) (int numsen,
          siginfo_t *datos, void *extra);
  sigset_t sa_mask;
  int sa_flags;
};
typedef struct {
  int si_signo;
  int si_code;
  union sigval si_value;
} siginfo_t;
union sigval {
  int sival_int;
  void *sival_ptr;
};
struct timespec {
        time_t tv_sec;
        long tv_nsec;
      };
struct itimerspec {
        struct timespec it_value;
        struct timespec it_interval;
      };
int sigaddset(sigset_t *pset, int sig); int sigdelset(sigset_t *pset, int sig);
int sigprocmask(int how, const sigset_t *set, siset *oset);  int pthread_setcanceltype(int type, int *oldtype);
type: PTHREAD_CANCEL_DEFERRED, PTHREAD_CANCEL_ASYNCRONOUS
int nanosleep(struct timespec *retraso, struct timespec *queda);
pid_t getpid(void); pid_t getppid(void); pid_t wait(int *estado); pid_t waitpid(pid_t, int *estado, int options);
mqd_t mq_open(const char *mq_name, int oflag, mode_t modo, struct mq_attr *atributos);
int mq_receive(mqd_t cola, const char *datos, size_t longitud, unsigned int *prioridad);
int mq_close(mqd_t cola); int mq_unlink(const char *nombre);
Modo: S_I + (R, W, X) + (USR, GRP, OTH). También S_IRWXU, S_IRWXG, S_IRWXO
Flags: O_RDONLY, O_WRONLY, O_RDWR, O_CREAT, O_EXCL, O_APPEND, O_TRUNC, O_NONBLOCK
int pthread_mutex_lock(pthread_mutex_t *mutex);  pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);  int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);  i
nt pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

Examen de Sistemas Informáticos en Tiempo Real
4º II (18/9/13)

PROBLEMA. TIEMPO: 1 HORA 30 MINUTOS (VALORACIÓN: 50%)

Resumen: Se pide realizar en C utilizando las normas POSIX el programa **reloj**, que al ejecutarse mantiene un **reloj interno y genera alarmas programables** para avisar a otros procesos. El programa **reloj** recibirá **órdenes** por medio de una cola de mensajes que permitirán poner en hora el reloj, leer la hora, programar las alarmas y parar. Algunas órdenes requieren una **respuesta** que se realizará mediante otra cola de mensajes distinta. Los macros, funciones y tipos mencionados **se suponen definidos** en la cabecera **reloj.h** (sólo hay que incluirla).

Especificación detallada:

- **Argumentos de la línea de comandos**:
  - **Argumento 1:** Señal de alarma enviada a otros procesos, **sig_aviso** (entero codificado en caracteres).
  - **Argumento 2:** Nombre de la cola de entrada por la que se reciben órdenes.
  - **Argumento 3:** Nombre de la cola de salida por la que se envían respuestas.
- **Creación de colas de mensajes:** El programa reloj crea las dos colas de mensajes con capacidad de mensajes **N_MSG** en ambos casos, y tamaño suficiente para una estructura **orden** (cola de entrada) o **respuesta** (cola de salida).
- **Lectura y ejecución de órdenes:** Los mensajes de la cola de entrada se interpretan como estructuras **orden**. Hay cuatro órdenes distintas (campo **tipo**):
  - PUESTA_EN_HORA: La hora recibida en el mensaje (campo **hora_deseada**) sustituye a la hora actual. No necesita respuesta.
  - PEDIR_HORA: Se envía por la cola de salida un mensaje que contiene una estructura de **respuesta** con la hora actual y **tipo** ACEPTADA.
  - PROGRAMAR_ALARMA: Si el número de alarmas pendientes no alcanza **N_MAX_ALARM** se programa una alarma nueva, y como se explica más adelante el proceso **id** recibirá una señal **sig_aviso** cuando la hora actual sea igual al campo **hora_deseada**. En cualquier caso se envía una respuesta, con tipo ACEPTADA (si se ha programado la alarma) o RECHAZADA (si no se ha programado).
  - PARAR: Para el proceso **reloj**, después de cerrar y destruir las colas de mensajes. No necesita respuesta.
- **Actualización del reloj:** El programa mantendrá cuentas de horas, minutos, segundos y décimas de segundo, independientes de la hora del reloj del sistema. La actualización se realizará por medio de un **temporizador POSIX 1003.1b**.
- **Alarmas:** Cuando la hora actual es igual a la programada para una de las alarmas, **reloj** envía al proceso destinatario una señal **sig_aviso** y desactiva la alarma, de modo que el número de alarmas pendientes disminuye.
- **Condiciones de fin:** El proceso **reloj** acaba al recibir una orden de tipo PARAR.

Otras condiciones y observaciones:

- El ciclo debe realizarse con un **temporizador POSIX 1003.1b**.
- Se recomienda utilizar **al menos dos hilos**, uno para atender la cola de órdenes y otro para actualizar el reloj, aunque se aceptan otras soluciones.
- El programa debe tener una representación interna de la hora, y una tabla de alarmas con indicación de si están o no pendientes.
- El programa deberá funcionar **independientemente** de los valores concretos de los argumentos y las constantes simbólicas.
- Es necesario utilizar mutex y variables de condición cuando la solución lo requiera, según lo indicado en clase.
- No es necesario considerar tratamiento de errores. Puede suponerse que tanto los argumentos de la línea de comandos como las órdenes recibidas son correctos.
- Es preciso acompañar el programa de pseudocódigo o explicación de su funcionamiento.

Fichero de cabecera reloj.h:

```
/* Cabecera reloj.h */
/* Capacidad de las colas */
#define N_MSG 4
/* Número máx. de alarmas pendientes */
#define N_MAX_ALARM 4

/* Tipos de petición recibidos */
enum petic {
    PUESTA_EN_HORA,
    PEDIR_HORA,
    PROGRAMAR_ALARMA,
    PARAR
};

/* Estructura que contiene una hora
   determinada */
struct hora {
    int horas;
    int minutos;
    int segundos;
    int decimas;
};

/* Peticiones recibidas */
struct peticion {
    enum petic tipo;
    struct hora hora_deseada;
    pid_t id;
};

/* Tipos de respuesta */
enum resp {
    ACEPTADA,
    RECHAZADA
}

/* Respuestas enviadas */
struct respuesta {
    enum resp tipo;
    struct hora hora_actual;
};
```