Examen de Sistemas Informáticos en Tiempo Real
IAEI-II(11/9/08)

CUESTIONES. TIEMPO: 1 HORA 15 MINUTOS (VALORACIÓN: 50%).

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

1. Explique cómo funciona la pila del sistema y para qué se utiliza normalmente.
2. 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. Explique también cómo podría implementarse utilizando las normas POSIX. Para todas las actividades el tiempo límite ("deadline") es igual al periodo.

Nombre	Periodo (ms)	Coste (ms)
A	20	3
B	30	4
C	30	5
D	60	12

3. Para el siguiente programa a) Explique **en general** su funcionamiento, suponiendo que recibe señales entre SIGRTMIN y SIGRTMIN+4, ambas inclusive, acompañadas de datos. b) Suponiendo que los efectos de una señal siempre tienen lugar antes de que llegue la siguiente, explique qué sucede y qué mensajes aparecen en pantalla cuando se envían al proceso que empieza en **main** todas las señales que se indican a continuación:
   - SIGRTMIN cada 100 ms desde t = 100, con dato 2.
   - SIGRTMIN + 4 en t = 150 con dato 0 y en t = 350 con dato 1.

```
<cabeceras correctas>
#define N1 5
#define N2 4
int t[N2];
void g(int a, siginfo_t *b, void *c) {
  t[a-SIGRTMIN] += b->si_value.sival_int;
}
int main(int argc, char **argv) {
  struct sigaction acc;
  int i, j, k, m; sigset_t d, d1; siginfo_t f;
  for(i=0; i<N2; i++) t[i] = 0;
  acc.sa_sigaction = g;
  acc.sa_flags = SA_SIGINFO;
  sigemptyset(&acc.sa_mask);
  sigemptyset(&d);
  for(i=SIGRTMIN; i<SIGRTMIN+N2; i++) {
    sigaction(i, &acc, NULL); sigaddset(&d, i);
  }
  sigprocmask(SIG_UNBLOCK, &d, NULL);
  sigemptyset(&d1);
  sigaddset(&d1, SIGRTMIN+N2);
  sigprocmask(SIG_BLOCK, &d1, NULL);
  do
  { j = sigwaitinfo(&d1, &f);
    if(j != -1) {
      if(f.si_value.sival_int == 0)
            k = SIG_BLOCK;
      else  k = SIG_UNBLOCK;
      printf("k vale %d\n", k);
      sigprocmask(k, &d, NULL);
    } else {
      m = 0;
      for(i=0; i<N2; i++) m+=t[i];
      printf("m vale %d\n", m);
    }
  } while(m < N1);
}
```

4. En la aplicación que describe este pseudocódigo la función **f1** debe ejecutarse cada 20 ms, **independientemente** de que existan o no mensajes pendientes de recibir en la cola de mensajes **cola1**. Explique al menos dos maneras de resolver el acceso a la cola de mensajes utilizando las normas POSIX. El pseudocódigo no debe cambiar; sólo su desarrollo.

```
<inicialización>
Mientras fin != 1
  Esperar siguiente ciclo de 20 ms
  Si hay mensajes en la cola cola1,
    leer mensaje de cola
      arg2 = f2(mensaje)
    Fin si
  fin = f1(arg1, arg2);
  Fin Mientras
  <más sentencias>
```

5. Explique el funcionamiento del siguiente programa, y cómo se modifica si se cambian las llamadas **pthread_cond_broadcast** por llamadas a **pthread_cond_signal**. Puede suponerse que todas las operaciones tardan un tiempo muy pequeño, salvo las que implican una espera.

```
<cabeceras correctas>
#define N 4
pthread_mutex_t m =
    PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c =
    PTHREAD_COND_INITIALIZER;
int n = 0;
void *f(void *p)  {
  struct timespec v = {0, 250000000L};
  int j = 0; int i = (int)p;
  pthread_mutex_lock(&m);
  while(n!=i) {
    pthread_cond_wait(&c, &m);
    j++;
  }
  n++;
  pthread_mutex_unlock(&m);
  nanosleep(&v, NULL);
  pthread_mutex_lock(&m);
  pthread_cond_broadcast(&c);
  pthread_mutex_unlock(&m);
  printf("Para %d j vale %d\n", i, j);
  return NULL;
}
int main(int argc, char **argv) {
  int i; pthread_t h[N];
  for(i=0; i<N; i++) {
  pthread_create(h+i, NULL, f, (void *)i); }
  pthread_mutex_lock(&m);
  n++;
  pthread_cond_broadcast(&c);
  pthread_mutex_unlock(&m);
  for(i=0; i<N; i++) {
    pthread_join(h[i], NULL);
    printf("Numero %d\n", i);
  }
  return 0;
}
```

Datos que pueden ser útiles:

```
int sigaction(int sig, struct sigaction *act, struct sigaction *oact);
int kill(pid_t pid, int sig);
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_receive(mqd_t cola, const char *datos, size_t longitud, unsigned int *prioridad);
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;
      };

int sigaddset(sigset_t *pset, int sig); int sigdelset(sigset_t *pset, int sig);
int sigprocmask(int how, const sigset_t *set, sigset *oset);
int sigqueue(pid_t pid, int sig, const union sigval val);
int mq_notify(mqd_t cola, const struct sigevent *espec); pid_t fork(void);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
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);
int mq_getattr(mqd_t cola, struct mq_attr *atributos);
int mq_close(mqd_t cola); int mq_unlink(const char *nombre);
int mq_notify(mqd_t cola, const struct sigevent *espec);
int timer_create(clockid_t reloj, struct sigevent *aviso, timer_t *tempo);
mqd_t mq_open(const char *mq_name, int oflag, mode_t modo, struct mq_attr *atributos);
Flags: O_RDONLY, O_WRONLY, O_RDWR, O_CREAT, O_EXCL, O_APPEND, O_TRUNC,
O_NONBLOCK
```

Examen de Sistemas Informáticos en Tiempo Real
IAEI-II(11/9/08)

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

Resumen: Se pide realizar en C y con llamadas POSIX un programa para crear el proceso A de la figura. El proceso A arranca al proceso B y recibe mensajes de él por medio de una cola POSIX.

- **Funcionamiento general del proceso A:** El programa que se utiliza para crear el proceso A recibe como **argumento 1** el nombre del ejecutable del proceso B, y como **argumento 2** el nombre que debe dar a la cola de mensajes. El proceso A **debe crear la cola**, con mensajes de tamaño **int** y capacidad de **LCOLA** mensajes. Una vez ha realizado las operaciones de inicialización el proceso A arranca el proceso B; si se incumplen las restricciones temporales es posible que B tenga que ser arrancado de nuevo una o varias veces, y cuando el proceso B acaba correctamente (sin ser abortado) el proceso A también debe terminar. Estos detalles se describen en el siguiente punto. Cada vez que arranca, el proceso B deberá recibir como **argumento 1** el nombre de la cola de mensajes.
- **Lectura de la cola y acciones sobre el proceso B:** El proceso A lee de la cola repetidamente mientras el proceso B no haya acabado correctamente; en cada recepción son relevantes tanto el dato recibido como el tiempo de espera:
    - **Tiempo de espera:** Si el mensaje que se recibe llega **menos de 20 o más de 60 milisegundos** después del instante en que se recibió el anterior, se considera que se ha producido un error de funcionamiento del proceso B, y el proceso A debe enviarle una señal SIGRTMIN para notificarselo. Se toleran un máximo de **N_MAX_ERR** errores de funcionamiento; si se alcanza este máximo en lugar de enviar SIGRTMIN el proceso A aborta el proceso B enviándole una señal SIGTERM, y después espera a que termine. También se aborta del mismo modo el proceso B si el mensaje **no ha llegado en 100 milisegundos**, sin esperar a que se produzca la recepción. En cualquier caso, después de abortar el proceso B éste ha de arrancarse de nuevo, y su cuenta de errores pasa a cero. Para el caso especial del primer mensaje puede tomarse como tiempo de espera el tiempo transcurrido desde el que comenzó a ejecutarse el proceso B hasta que se produce la recepción.
    - **Dato recibido:** Si el entero recibido vale **ACTIVO**, entonces el proceso B va a seguir funcionando, y el proceso A debe seguir recibiendo mensajes. Si vale **FIN**, el proceso B está acabando, y ya no enviará más mensajes; este caso es la **condición de fin del proceso A**, que deberá esperar a que B acabe realmente, y terminar después.

NOTAS:

- Es necesario utilizar uno o varios **temporizadores POSIX** (**no retrasos**) cuando sean necesaria su funcionalidad para controlar el intervalo de tiempo entre mensajes.
- Las constantes simbólicas se encuentran en la cabecera **proceso.h**.
- No es necesario considerar tratamiento de errores en las llamadas al sistema.
- Es preciso acompañar el programa de pseudocódigo o explicación de su funcionamiento.

```
/* proceso.h */

#define LCOLA          5    /* Capacidad de la cola */
#define N_MAX_ERR      5    /* Numero maximo de errores */
#define ACTIVO         0    /* Proceso B activo */
#define FIN            1    /* Proceso B terminando */
```