Examen de Sistemas Informáticos en Tiempo Real
4º II (8/6/12)

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. Para el siguiente programa explique **en general** cómo funciona, cuánto tarda en ejecutarse y qué mensajes imprime por pantalla, suponiendo que ninguno de los procesos creados recibe señales aparte de las que puedan enviarse entre ellos. En particular, explique qué sucede cuando el programa se ejecuta con el comando "**c1 1 0 2**", siendo "**c1**" el nombre del ejecutable.

```
<cabeceras correctas>
int a;
void m(int s, siginfo_t *p, void *p1) {
  a += (s – SIGRTMIN);
}
void f(int i, int n) {
  struct timespec t = {0, 500000000L};
  while(n--) {
    kill(getppid(), SIGRTMIN+i);
    nanosleep(&t, NULL);
  }
  exit(0);
}
int main(int b, char **c) {
  int i, k, n; pid_t p;
  sigset_t v1; struct sigaction s;
  s.sa_sigaction = m;
  s.sa_flags = SA_SIGINFO;
  sigemptyset(&s.sa_mask);
  for(i=0; i<b; i++)  {
    sigemptyset(&v1); sigaddset(&v1, SIGRTMIN+i);
    sigaction(SIGRTMIN+i, &s, NULL);
  }
  sigprocmask(SIG_UNBLOCK, &v1, NULL);
  for(i=1; i<b; i++) {
    sscanf(c[i], "%d", &n);
    p = fork(); if(p == 0) f(i, n);
  }
  for(i=1; i<b; i++) {
    while(wait(&k) == -1);
  }
  printf("valor %d\n", a);
  return 0;
}
```

2. Para el siguiente programa explique **en general** cómo funciona, cuánto tarda en ejecutarse y qué mensajes imprime por pantalla. En particular, explique qué sucede cuando el programa se ejecuta a) con el comando "**c3**" b) con el comando "**c3 3**"**.** En ambos casos "**c3**" es el nombre del ejecutable.

```
<cabeceras correctas>
#define N 5
pthread_cond_t c =
  PTHREAD_COND_INITIALIZER;
pthread_mutex_t m =
  PTHREAD_MUTEX_INITIALIZER;
int a, b, d;
void *f1(void *p) {
  int res;
  pthread_mutex_lock(&m);
  while(a == b) {
    pthread_cond_wait(&c, &m);
  }
  d--;  printf("d vale %d\n", d);
  pthread_mutex_unlock(&m);
}
int main(int v1, char **v2) {
  int i, k, n;  pthread_t h[N];
  struct timespec t = {1, 0L};
  a = b = 0;
  for(i = 0, d = 0; i<N; i++) {
    pthread_create(&h[i], NULL, f1, NULL);
    d++;
  }
  if(v1>=2) sscanf(v2[1], "%d", &k);
  else k = 0;
  n = 1;
  while(n) {
    printf("Mensaje 1\n");
    pthread_mutex_lock(&m);
    a++;
    if(a != b)  {
      if(a < k) pthread_cond_signal(&c);
      else pthread_cond_broadcast(&c);
    }
    pthread_mutex_unlock(&m);
    nanosleep(&t, NULL);
    pthread_mutex_lock(&m);
    if(d == 0) n = 0;
    pthread_mutex_unlock(&m);
  }
  return 0;
}
```

3. Explique a) por qué aparece y en qué consiste el problema de la reubicación **después** de crear un ejecutable b) en qué consiste el direccionamiento virtual y paginado c) cómo puede utilizarse este tipo de direccionamiento para solucionar el problema de reubicación del apartado a).
4. Explique cómo funcionan estos dos fragmentos de programa en ADA, a qué tipo de sincronización corresponden y cómo podría conseguirse una funcionalidad parecida utilizando colas de mensajes POSIX.

```
<sentencias de la tarea tar>
  accept entrada1(I: in integer) do
  <Sentencias A>
  end entrada1;
<más sentencias de la tarea tar>
```

```
<sentencias de la tarea tar1>
tar.entrada1(7);
<más sentencias de la tarea tar1>
```

5. Explique el concepto de semáforo, y cómo puede utilizarse para resolver los dos tipos de sincronización explicados en clase.

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
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 (8/6/12)

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

Resumen: Se pide realizar en C y con llamadas POSIX el programa **bolsa** que recibe órdenes de compra y venta de acciones por una cola de mensajes POSIX 1003.1b y responde mediante señales de tiempo real a los procesos clientes que las han enviado para notificar si se han ejecutado correctamente o se han cancelado. Los tipos y las constantes simbólicas necesarios están definidos en la cabecera **bolsa.h**, que se supone disponible.

Especificación detallada:

- **Inicialización:** El programa crea la cola de recepción con el nombre definido por el **argumento 1** de la línea de comandos, mensajes de tamaño **struct operacion** y capacidad para **LCOLA_OP** mensajes. También debe crear una **tabla de órdenes pendientes**, con capacidad definida por el **argumento 2** de la línea de comandos (número máximo de órdenes pendientes; entero codificado en caracteres).
- **Recepción de órdenes:** El programa recibe estructuras **struct operacion** por la cola de mensajes. Cada vez que recibe una, busca una entrada libre en la tabla de órdenes pendientes. Si la hay almacena la orden; si no, la rechaza, respondiendo al proceso cliente con una señal **SIGRTMIN** y dato **OCUPADO**.
- **Ejecución de órdenes:** Las órdenes pendientes de la tabla indican que se quiere comprar o vender (**tipo_orden**) un cierto número de acciones (**numero**) de un determinado tipo (**codigo_accion**) a un **precio** (máximo si se compra, mínimo si se vende) y caducan al cabo de un cierto tiempo (**tiempo_limite**, en milisegundos). El campo **importe** está inicialmente a cero. Cada **tcic** milisegundos el programa analiza todas las órdenes pendientes. El valor **tcic** está definido por el **argumento 3** de la línea de comandos (milisegundos; entero codificado en caracteres).
    - Si se ha **agotado el tiempo límite** de la orden y no se ha comprado o vendido ninguna acción, la orden se anula y se envía al proceso cliente una señal **SIGRTMIN** con dato **CADUCADA**. Las órdenes parcialmente ejecutadas no caducan.
    - Si la orden es de compra, se buscan en la tabla órdenes pendientes de venta para **el mismo código de acción y con un precio menor o igual al de compra**. Cada vez que se encuentra una, se compran tantas acciones como sea posible hasta completar el total a comprar. En cada compra el precio es la **media de los dos precios** (el de compra y el de venta), y se actualizan adecuadamente los campos de las dos órdenes. El **importe** de las dos órdenes se incrementa en el coste de la compra o venta parcial (número de acciones compradas por el precio).
    - Cuando se han comprado o vendido todas las acciones de una orden, ésta se anula, y se envía al proceso cliente con una señal **SIGRTMIN** con el **importe total** de la compra o venta. Las órdenes pueden tardar varios ciclos en completarse.
- **Condiciones de parada:** El proceso para cuando la tabla no tiene órdenes pendientes durante al menos 10 minutos.

Otras condiciones:

- El ciclo debe realizarse con un **temporizador POSIX**.
- Se supone que todos los datos de las órdenes tienen sentido.
- El programa puede realizarse utilizando hilos o no. Si se utilizan, debe asegurarse en todo momento la coherencia de los datos por los medios adecuados.
- No es necesario considerar tratamiento de errores.
- Es preciso acompañar el programa de pseudocódigo o explicación de su funcionamiento.

Fichero de cabecera:

```
/* Cabecera bolsa.h */

/* Tipos de ordenes */
#define COMPRA 0  /* Compra de acciones */
#define VENTA  1  /* Venta de acciones */
#define NULA   2  /* Tipo para marcar la orden como nula */

struct operacion
{
   int tipo_orden;      /* Se recibe COMPRA o VENTA */
   pid_t proceso;       /* pid del proceso que envía la orden */
   int codigo_accion;   /* Código de la acción a comprar o vender */
   int numero;          /* Numero de acciones  a comprar o vender */
   int precio;          /* Precio en céntimos (mínimo para vender y máximo para comprar) */
   int tiempo_limite;   /* Máximo tiempo de actividad de la orden (ms) */
   int importe;         /* Importe total en céntimos de la venta o compra (inicialmente a cero) */
};

/* Respuestas */
#define OCUPADO   -1  /* No hay lugar en la tabla de órdenes pendientes */
#define CADUCADA -2  /* Tiempo límite superado */

/* Longitud de la cola */
#define LCOLA_OP 10
```