Examen de Informática Industrial
4º GITI (4/6/14)

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

1. Explique a) Qué es un proceso, y qué estructuras de datos lo componen. b) Qué es un hilo y cuáles son sus diferencias con respecto a un proceso. c) Ventajas e inconvenientes de ambos.
2. 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** recibe señales SIGRTMIN acompañadas de datos positivos. En particular, explique qué sucede cuando dicho proceso recibe señales SIGRTMIN con datos 1, 1, 3, 3, 3, 4, 3, 2 y 5 a los 100, 200, 300, 600, 700, 800, 1200, 1300 y 1400 milisegundos desde el comienzo y se invoca como a) **c2 5 4 5** y b) **c2 6**.

```
<cabeceras correctas>
#define N 3
pid_t p[N];
void f1(int i, int c){
  int j = 0, k; sigset_t s; siginfo_t x;
  struct timespec t = {0, 500000000L};
  struct itimerspec t2; timer_t b;
  sigemptyset(&s); sigaddset(&s, SIGRTMIN);
  sigaddset(&s, SIGALRM);
  sigprocmask(SIG_BLOCK, &s, NULL);
  timer_create(CLOCK_REALTIME,
               NULL, &b);
  t2.it_value = t; t2.it_interval = t;
  timer_settime(b, 0, &t2, NULL);
  while(j < c) {
    k = sigwaitinfo(&s, &x);
    if(k == SIGALRM) j = 0;
    else j = j + x.si_value.sival_int;
    printf("%d con %d\n", i, j);
  }
  printf("%d tras while 1\n", i);
  if(i == 0) return;
  do {
    k = sigwaitinfo(&s, &x);
    if(k==SIGRTMIN)
      sigqueue(p[i-1], SIGRTMIN, x.si_value);
  } while(x.si_value.sival_int >= 0);
  printf("%d tras while 2\n", i);
}
int main(int b, char **b1){
  pid_t p1; int i, k, c; sigset_t v; siginfo_t a;
  struct sigaction h; struct timespec t = {0, 30000000};
  sscanf(b1[1], "%d", &c);
  h.sa_flags = SA_SIGINFO; h.sa_sigaction = f2;
  sigemptyset(&h.sa_mask);
  sigaction(SIGRTMIN, &h, NULL);
  sigemptyset(&v); sigaddset(&v, SIGRTMIN);
  sigprocmask(SIG_BLOCK, &v, NULL);
  for(i=0; i<N; i++) {
    p[i] = fork();
    if(p[i] == 0) {
      f1(i, c); exit(0);
    }
  }
  do {
    sigwaitinfo(&v, &a);
    sigqueue(p[N-1], SIGRTMIN, a.si_value);
    nanosleep(&t, NULL);
    p1 = waitpid(p[0], &k, WNOHANG);
  } while(p1 == 0);
  a.si_value.sival_int = -1;
  sigqueue(p[N-1], SIGRTMIN, a.si_value);
  return 0;
}
```

3. 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	16	2
B	16	3
C	24	10
D	48	5

4. Explique las particularidades del tratamiento de señales POSIX cuando existen varios hilos en un proceso.
5. Para el siguiente programa explique **en general** cómo **funciona**, cuánto **tarda** en ejecutarse y qué **mensajes** imprime por pantalla suponiendo que recibe señales SIGRTMAX acompañadas de datos positivos diferentes entre sí. En particular explique qué sucede cuando
   - Se invoca como **c5 1 1 2 3** y recibe señales SIGRTMAX con datos 1, 3, 4 y 2 a los 50, 100, 150 y 200 ms de comenzar.
   - Otros procesos envían mensajes con datos 1, 1, 1, 3, 2, 1 y 1 a los puertos UDP 10000, 10002, 10001, 10001 10004, 10003 y 10001 respectivamente, el primero a los 400 ms de comenzar y el resto cada 200 ms.

```
<cabeceras correctas>
#define N 3
#define D 10000
int *d; int n; pthread_t v[N];
void *f1(void *p) {
  int i, j, k, m, g, h; struct sockaddr_in c;
  j = (int)p;
  k = socket(AF_INET, SOCK_DGRAM, 0);
  memset((char *)&c, 0, sizeof(c));
  c.sin_family = AF_INET;
  c.sin_port = htons(D+j);
  c.sin_addr.s_addr = INADDR_ANY;
  bind(k, (struct sockaddr *)&c, sizeof(c));
  do {
    recv(k, (void *)&m, sizeof(m),
         MSG_WAITALL);
    for(i=0, h=0; i<n; i++) {
      if(d[i] == m) {
        d[i] = 0; h = 1;
        printf("%d: mensaje 1\n", j);
      }
    }
    for(i=0, g=0; i<n; i++)
      if(d[i] != 0) g++;
  } while(g>0);
  for(i=0; i<N; i++) pthread_cancel(v[i]);
  return NULL;
}
void f2(int a, siginfo_t *b, void *c) { }
int main(int a, char **b){
  int i, j; sigset_t s; siginfo_t t;  struct sigaction h;
  d = (int *)malloc(sizeof(int)*(a-1));
  for(i=1; i<a; i++)
    sscanf(b[i], "%d", &(d[i-1]));
  n = a-1;
  h.sa_flags = SA_SIGINFO; h.sa_sigaction = f2;
  sigemptyset(&h.sa_mask);
  sigaction(SIGRTMAX, &h, NULL);
  sigemptyset(&s); sigaddset(&s, SIGRTMAX);
  pthread_sigmask(SIG_BLOCK, &s, NULL);
  for(i=0; i< N; i++) {
    sigwaitinfo(&s, &t);
    j = t.si_value.sival_int;
    pthread_create(&v[i], NULL, f1, (void *)j);
  }
  for(j=0; j<N; j++) pthread_join(v[j], NULL);
  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);
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_cancel(pthread_t thread);
int sigwaitinfo(const sigset_t *estas_sg, siginfo_t *infop);
int sscanf(char  *fuente, char *formato, …);  int sprintf(char *destino, char *formato,…);
int pthread_sigmask(int how, const sigset_t *set, sigset_t *oset); int sigemptyset(sigset_t *pset);
int sigfillset(sigset_t *pset); int sigaddset(sigset_t *pset, int sig); int sigdelset(sigset_t *pset, int sig);
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 sockaddr_in {
  sa_family_t    sin_family;
  in_port_t      sin_port;
  struct in_addr sin_addr;
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 pthread_mutex_lock(pthread_mutex_t *mutex); int pthread_mutex_unlock(pthread_mutex_t *mutex);
int socket(int dominio, int tipo, int protocolo); ssize_t recv(int id_socket, void *buffer, size_t tam, int flagsr);
int bind(int socket, const struct sockaddr *direccion, socklen_t long_dir);
ssize_t sendto(int id_socket, void *pdatos, size_t tam, int flagss, struct sockaddr *destino, socklen_t tam_dest);
uint32_t htonl(uint32_t local); uint16_t htons(uint16_t local);
int listen(int socket, int tam_cola); int accept(int socket, struct sockaddr *direccion, socklen_t *lon_dir);
int connect(int socket, const struct sockaddr *direccion, socklen_t tam);
```

Examen de Informática Industrial
4º GITI (4/6/14)

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

Resumen: Se pide realizar en C y con llamadas POSIX un programa para crear un proceso multihilo que controla el sistema de vehículos de la figura. El proceso tendrá al menos un hilo de lectura de órdenes y **N_VEH** hilos de control de vehículo; utiliza un "socket" orientado a conexión para recibir órdenes de carga de varios tipos de piezas y enviar resultados para cada orden recibida. Los macros, funciones y tipos mencionados **se suponen definidos** en la cabecera **medidas.h**, que se supone disponible (sólo hay que incluirla).

Especificación detallada:

- **Argumentos de la línea de comandos:**
  - **Argumento 1:** Define **npuerto** (entero codificado en caracteres; puerto TCP).
  - **Argumento 2:** Dirección remota para conectar el "socket" en formato de cadena de caracteres (códigos decimales separados por puntos).
  - **Argumento 3:** Define **tmax** (entero codificado en caracteres; tiempo en segundos).
- **Conexión y datos del "socket"**: El "socket" podrá vincularse a cualquier puerto TCP disponible y se conectará activamente al puerto TCP **npuerto** de la dirección IP definida por el argumento 2. Los datos que llegan al "socket" contienen estructuras **struct msg_orden**. Los datos enviados deben contener estructuras **struct msg_res**. Cada mensaje de resultado debe contener el identificador de la orden correspondiente, **id_orden** (ver cabecera "**problema.h**"). Se supone que no hay problemas con la ordenación de los octetos.
- **Posiciones y pasillo**: Hay **N_VEH** posiciones de aparcamiento y descarga, una por vehículo, y **N_TIPO** posiciones de carga, una por tipo de pieza. Para ir de una posición a otra el vehículo siempre tiene que pasar por el pasillo. Para evitar colisiones **no puede haber más de un vehículo a la vez** en el pasillo y en cada una de las posiciones de carga.
- **Hilo de lectura de órdenes:** Este hilo recibe órdenes por la cola de recepción de órdenes y las copia en los datos compartidos para que algún vehículo las acepte. El hilo de lectura de órdenes tendrá que **esperar a que la orden anterior haya sido aceptada** antes de copiar la siguiente. Si trascurren **más de tmax** segundos antes de que la nueva orden pueda copiarse, el hilo de lectura renuncia a ejecutarla, envía un mensaje con resultado **RES_ERROR** por el "socket" y pasa a recibir la siguiente.
- **Hilos de control de vehículos:** Existirán **N_VEH** hilos de vehículo con código idéntico, aunque controlan vehículos distintos.
  - Cada hilo de control **espera a que exista una nueva orden** en los datos compartidos; cuando consigue el acceso a una nueva orden la convierte en orden aceptada y comienza a ejecutarla.
  - Para ejecutar una orden el vehículo tiene que dirigirse sucesivamente a cada uno de los **N_TIPO** puntos de carga desde su punto de aparcamiento, cargar el número de piezas que indica la orden para cada tipo (tabla **np** de **struct msg_orden**), volver al punto de aparcamiento y descargar. Si el número de piezas de algún tipo es cero, pasa directamente al siguiente.
  - Para moverse a una posición de carga tiene que **esperar a que tanto ésta como el pasillo estén libres**, porque no puede haber dos vehículos a la vez en ninguno de los dos lugares.
  - Cuando el vehículo ha acabado de cargar y descargar todas las piezas necesarias **envía un mensaje por el "socket"** con resultado **RES_OK** y vuelve a esperar otra orden.
  - Para realizar todas estas operaciones pueden utilizarse las funciones **carga**, **descarga** y **mueve** (ver cabecera "**problema.h"**), que se suponen existentes. Inicialmente puede suponerse que todos los vehículos están en sus puntos de aparcamiento.

Otras condiciones y observaciones:

- Se supone que todas las llamadas POSIX son "**thread-safe**".
- El programa deberá funcionar **independientemente** de los valores concretos de los argumentos y las constantes simbólicas.
- **Utilice mutex y variables de condición** cuando sean necesarios para gestionar las variables compartidas según lo indicado en clase, evitando (cuando sea posible) permanecer un tiempo largo en la sección crítica.
- No existe condición de fin.
- No es necesario considerar tratamiento de errores. Puede suponerse que los argumentos de la línea de comandos son correctos.
- Es preciso acompañar el programa de pseudocódigo o explicación de su funcionamiento.

```
/* Cabecera problema.h */
#define N_VEH       3
#define N_TIPO      3
#define TMAX       10
#define FILA_AP     0
#define FILA_PAS    1
#define FILA_ALM    2
#define COL_VEH0    0
#define COL_ALM0    0

/* Cargar n piezas de tipo id_tipo */
void cargar(int id_tipo, int n);

/* Descargar vehículo idv */
void descargar(int idv);

/* Mover veh. idv a fila y columna */
void mueve(int idv, float fila, float col);

/* Mensaje de orden */
struct msg_orden{
   int id_orden;     /* Id de la orden,
                        siempre >0 */
   int np[N_TIPO];  /* Num. Piezas a cargar
                       para cada tipo*/
};

/* Mensaje de respuesta */
#define RES_OK      0 /*Orden ejecutada*/
#define RES_ERROR 1 /*Error, no ejecutada*/
struct msg_respuesta {
  int id_orden;  /* Id de la orden*/
  int resultado; /* Resultado */
};
```