PRACTICA 3 Sistema de votación multiproceso

Sara Serrano Marazuela y Xiomara Caballero Cuya

INTRODUCCIÓN

En esta práctica, el objetivo principal es crear un sistema de votación multiproceso, donde tendremos un candidato y diferentes votantes que decidirán si eligen o no la propuesta.

Para ello, tendremos un proceso Principal que se encargará de crear múltiples procesos Votante (según el número especificado en los argumentos) y de entre esos, se elegirá a un Candidato. El valor que se introduce por argumentos es el número de Votantes que va a haber, por lo que tendremos que crear un proceso más que será el Candidato.

Finalmente, se repetirán diferentes rondas, que tan solo termirán cuando llegue la señal SIGINT o cuando termine la alarma (parámetro que también pasaremos por argumentos).

Para conseguir estos objetivos hemos utilizado diferentes semáforos y señales entre otros recursos.

La funcionalidad de los diferentes semáforos se explica a continuación:

- **Semáforo 1**: nos permite distinguir cuándo ya se ha elegido un Candidato para que el resto de procesos figuren como Votantes.
- **Semáforo 2**: evita que dos procesos escriban en el fichero al mismo tiempo
- **Semáforo 3**: usado para saber cuándo los procesos votante han terminado de votar y de escribir en el fichero

Además, se han usado las siguientes señales:

- **SIGINT**: terminar el programa y las rondas
- **SIGALRM**: terminar el programa por medio de la alarma en función del tiempo que se

ha establecido

- **SIGTERM**: liberar los recursos una vez que ha terminado el programa
- **SIGUSR1**: señal que se envía a los votantes cuando principal está listo y que se envía para comenzar una nueva ronda
- **SIGUSR2:** señal que se envía para indicar que los procesos Votante pueden comenzar a registrar su voto

Además, para todas estas señales hemos definido las funciones que se utilizarán para recibir la señal, además de definir las estructuras necesarias para su correcto uso.

CÓDIGO PRINCIPAL

Para realizar esta práctica hemos creado dos ficheros, principal y votante. En principal.c declaramos las funciones encargadas de gestionar las señales y el main que ejecuta el programa.

HANDLES

Para el handle de **SIGINT** lo que hace es mandar una señal **SIGTERM** a todos los procesos para que liberen sus recursos. Para ello, debe leer el fichero para acceder a los pids. Por último, libera los semáforos utilizados e imprime el mensaje por pantalla "Finishing by signal".

```
void handle_sigint(int sig) {
    (void)sig;
    int fd, num_procesos=0, *pids=NULL;
    int i;

    fd = open("PIDS.txt", O_RDONLY);
    if (fd == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }


    pids = readpids(fd, &num_procesos);
    if(pids == NULL) {
        perror("readpids");
        exit(EXIT_FAILURE);
```

```
    }

    if(getpid() == pids[0]) {
        printf("Finishing by signal\n");
    }
    fflush(stdout);
    usleep(250000);

    close(fd);  // Cerrar el fichero

    for (i = 0; i < num_procesos+1; i++) {
        if (kill(pids[i], SIGTERM) == -1) {
            perror("kill");
            free(pids);
            exit(EXIT_FAILURE);
        }
    }

    for (i = 0; i < num_procesos+1; i++) {
        wait(NULL);
    }

    free(pids);

    unlink(FICHERO);
    sem_unlink("/sem1");
    sem_unlink("/sem2");
    sem_unlink("/sem3");

    //unlink(FICHERO);
    exit(EXIT_SUCCESS);
}
```

Por otro lado, tenemos el handle de **SIGALRM** que se activa cuando pasan los segundos

pasados al ejecutar el programa. Esta fucnión se encarga de liberar los procesos de la misma forma que handle_sigint. Además imprime por pantalla el mensaje “Finishing by alarm”.

```
void handle_sigalarm(int sig) {
    (void)sig;
    int fd, num_procesos=0, *pids=NULL;
    int i;
    printf("Finishing by alarm\n");

    fd = open("PIDS.txt", O_RDONLY);
    if (fd == -1) {
        perror("Error al abrir el archivo de PIDs");
        exit(EXIT_FAILURE);
    }

    pids = readpids(fd, &num_procesos);
    if(pids == NULL) {
        perror("readpids");
        exit(EXIT_FAILURE);
    }

    close(fd);  // Cerrar el fichero

    // Enviar SIGTERM a cada proceso votante
    for (i = 0; i < num_procesos; i++) {
        if (kill(pids[i], SIGTERM) == -1) {
            perror("kill");
            free(pids);
            exit(EXIT_FAILURE);
        }
    }


    sem_unlink("/sem1");
    sem_unlink("/sem2");
    sem_unlink("/sem3");
    unlink(FICHERO);
    exit(EXIT_SUCCESS);
```

```
}


void handle_sigusr1(int sig) {
    (void)sig;
}

void handle_sigusr2(int sig) {
    (void)sig;
}
```

MAIN

En esta función primero debemos comprobar los parámetros que recibimos y volver a pedirlos en caso de que no sean válidos. A continuación, asignamos a cada señal su correspondiente estructura para poder programar la alarma y que empiece a contar el tiempo. Luego, inicializamos los semáforos y tras crear los procesos (número de procesos votantes más uno que será el proceso candidato) escribimos en el fichero estos datos. Por último, esta función envía la señal a los procesos creados

```
int main (int argc, char *argv[]){

    sem_t *sem1 = NULL;
    sem_t *sem2 = NULL;
    sem_t *sem3 = NULL;
    int val2, nProc=0, nSec=0;

    // INICIALIZAR HANDLES
    int i, fd;
    pid_t pid;
    struct sigaction act_int, act_usr1, act_usr2, act_term, act_alarm;
    sigset_t mask;
    int pids[MAX_PID];
```

```
    if(argc==3) {
        nProc = atoi(argv[1]);
        nSec = atoi(argv[2]);
    }

    // CONTROL DE PARÁMETROS
    if (argc != 3 || nProc < 1 || nSec < 1 || nProc >= MAX_PID) {
        while(nProc < 1 || nSec < 1|| nProc >=  MAX_PID || ) {


            printf("Error   al   introducir   los   parámetros,   vuelve   a
intentarlo.\n");
            if(nProc < 1 || nProc >= MAX_PID){
                printf("El número de votantes debe encontrar entre 1 y 29\n");
            }
            if(nSec < 1){
                printf("Los segundos deben ser mayor que 0\n");
            }

            scanf("%d %d", &nProc, &nSec);

        }
    }

    sigemptyset(&mask);
    sigaddset(&mask, SIGINT);
    sigaddset(&mask, SIGALRM);
    sigaddset(&mask, SIGTERM);
    sigaddset(&mask, SIGUSR1);
    sigaddset(&mask, SIGUSR2);

    act_int.sa_handler = handle_sigint;
    sigemptyset(&(act_int.sa_mask));
    act_int.sa_flags = 0;

    act_usr1.sa_handler = handle_sigusr1;
```

```
    sigemptyset(&(act_usr1.sa_mask));
    act_usr1.sa_flags = 0;

    act_usr2.sa_handler = handle_sigusr2;
    sigemptyset(&(act_usr2.sa_mask));
    act_usr2.sa_flags = 0;

    act_term.sa_handler = handle_sigterm;
    sigemptyset(&(act_term.sa_mask));
    act_term.sa_flags = 0;

    act_alarm.sa_handler = handle_sigalarm;
    sigemptyset(&(act_alarm.sa_mask));
    act_alarm.sa_flags = 0;

    if (sigaction(SIGINT, &act_int, NULL)<0) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }

    if (sigaction(SIGTERM, &act_term, NULL)<0) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }

    if (sigaction(SIGALRM, &act_alarm, NULL)<0) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }

    if (sigaction(SIGUSR1, &act_usr1, NULL)<0) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }

    if (sigaction(SIGUSR2, &act_usr2, NULL)<0) {
        perror("sigaction");
```

```
        exit(EXIT_FAILURE);
    }

    // ALARM
    alarm(nSec);

    sem_unlink("/sem1");
    sem_unlink("/sem2");
    sem_unlink("/sem3");

    // INICIALIZAR LOS SEMÁFOROS
    if((sem1 = sem_open("/sem1", O_CREAT, S_IRUSR | S_IWUSR, 0))==SEM_FAILED)
{
        perror("sem_open");
        exit(EXIT_FAILURE);
    }
    sem_post(sem1);

    if((sem2 = sem_open("/sem2", O_CREAT, S_IRUSR | S_IWUSR, 0))==SEM_FAILED)
{
        perror("sem_open");
        exit(EXIT_FAILURE);
    }
    sem_post(sem2);

    if((sem3    =    sem_open("/sem3",    O_CREAT,    S_IRUSR    |    S_IWUSR,
nProc))==SEM_FAILED) {
        perror("sem_open");
        exit(EXIT_FAILURE);
    }
    sem_getvalue(sem3, &val2);

    // ABRIR FICHERO CON LOS PIDS
    fd = open(FICHERO, O_CREAT | O_WRONLY | O_TRUNC, S_IRUSR | S_IWUSR);
    dprintf(fd, "%d\n", nProc);

    /// CREAR PROCESOS VOTANTES
```

```
    for (i = 0; i< nProc + 1; i++){
        pid= fork();
        if (pid < 0) {
            perror("fork");
            exit(EXIT_FAILURE);
        } else if (pid == 0) {
            votante(fd, sem1, sem2, sem3, nProc, mask);
            exit(EXIT_SUCCESS);
        } else {
            // Proceso principal guarda el PID del votante
            pids[i] = pid;
        }
    }

    // ESCRIBIR EN EL FICHERO LOS PID
    for (i = 0; i<nProc+1; i++){
        if(dprintf(fd, "%d ", pids[i]) == -1) {
            return EXIT_FAILURE;
        }
    }

    dprintf(fd, "\n");
    close(fd);

    // ENVIAR SEÑALES A VOTANTES
    for (i = 0; i < nProc+1; i++) {
        if(kill(pids[i], SIGUSR1) == -1) {
            perror("kill");
            return EXIT_FAILURE;
        }
    }

    while (!terminar);
}

    }
```

```
    while (!terminar);

    return status_total;
}
```

CÓDIGO VOTANTE

En este módulo tenemos cuatro funciones, readpids, readvotes, chooseCandidato y votante que se encargan de manejar a los procesos votantes y que impriman sus votos.

READPIDS

Esta función recibe como argumentos el fichero para leer, un puntero al número de procesos, para modificar su valor al leer el fichero. Primero lee el número de procesos que se encuentra en la primera fila y luego lee el valor de los pids de los procesos votantes y el proceso candidato que se encuentran separados por espacios.

```
int *readpids(int fd, int *nProc) {
    int i;

    int bytes_leidos;
    char buffer[1024], *token;

    int *pids = (int*)malloc(MAX_PID*sizeof(int));
    if (pids == NULL) {
        perror("malloc");
        return NULL;
    }

    fd = open(FICHERO, O_CREAT | O_RDONLY, S_IRUSR | S_IWUSR);
    if(fd == -1) {
        perror("open");
        free(pids);
        return NULL;
    }
    bytes_leidos = read(fd, buffer, sizeof(buffer) - 1);
    if(bytes_leidos == -1) {
        perror("read");
```

```
        free(pids);
        close(fd);
        return NULL;
    }
    buffer[bytes_leidos] = '\0';

    // Leer los pids
    token = strtok(buffer, "\n");
    *nProc = atoi(token);
    for(i= 0; i< *nProc +1; i++){
        token = strtok(NULL, " ");
        if(token == NULL) {
            perror("strtok");
            free(pids);
            close(fd);
            return NULL;
        }
        pids[i] = atoi(token);
        if(pids[i]<0) {
            perror("atoi");
            free(pids);
            close(fd);
            return NULL;
        }
    }

    close(fd);

    return pids;

}
```

READVOTES

Esta función recibe como parámetro el fichero para leer, el número de procesos y el número de ronda en la que se encuentra. Al igual en la función anterior, en la primera línea se lee el número

de procesos, en la segunda línea los pids de dichos procesos y por último se van creando tantas líneas como votos se hayan producido en los procesos votantes. Estos votos se van acumulando y escribiendo al final del fichero por lo que se deberá calcular que votos leer según la ronda en la que se encuentra.

```
char *readvotes(int fd, int *nProc, int *ronda) {
    int i, j, flag;

    int bytes_leidos;
    char buffer[1024], *token;

    int *pids = (int*)malloc(MAX_PID*sizeof(int));
    if (pids == NULL) {
        perror("malloc");
        return NULL;
    }

    char *votes = (char*)malloc(MAX_PID*sizeof(char));
    if(votes == NULL) {
        perror("malloc");
        free(pids);
        return NULL;
    }

    fd = open(FICHERO, O_CREAT | O_RDONLY, S_IRUSR | S_IWUSR);
    if(fd == -1) {
        perror("open");
        free(pids);
        free(votes);
        return NULL;
    }

    /*ronda = *ronda + 1;*/
    bytes_leidos = read(fd, buffer, sizeof(buffer) - 1);
    if(bytes_leidos == -1) {
        perror("read");
        close(fd);
        free(pids);
```

```
        free(votes);
        return NULL;
    }
    buffer[bytes_leidos] = '\0';

    // Leer los pids
    token = strtok(buffer, "\n");
    *nProc = atoi(token);
    for(i= 0; i< *nProc; i++){
        token = strtok(NULL, " ");
        pids[i] = atoi(token);
    }
    token = strtok(NULL, "\n");
    pids[i] = atoi(token);

    flag=1;
    while (flag==1) {
        flag=0;
        for(j=0; j < *ronda-1; j++){
            for (i = 0; i < *nProc && token != NULL; i++) {
                token = strtok(NULL, " ");  // Coge el pid
                token = strtok(NULL, " ");  // Coge la palabra "vota"
                token = strtok(NULL, "\n");  // Dividir por espacios
                votes[i] = token[0];  // Almacenar el voto (Y o N)
            }
        }
        for (i = 0; i < *nProc && token != NULL; i++) {
            token = strtok(NULL, " ");  // Coge el pid
            if(token == NULL) {
                flag = 1;
            } else {
                token = strtok(NULL, " ");  // Coge la palabra "vota"
                if (token == NULL) {
                    flag = 1;
                } else {
                    token = strtok(NULL, "\n");  // Dividir por espacios
                    if (token == NULL) {
```

```
                        flag=1;
                    } else {
                        votes[i] = token[0];  // Almacenar el voto (Y o N)
                    }
                }
            }
        }
    }

    free(pids);
    close(fd);
    return votes;

}
```

CHOOSECANDIDATO

Esta función recibe por parámetros los semáforos que se van a utilizar, en este caso 3, el número de procesos, la máscara de señales que debe bloquear y el número de ronda que está ejecutando.

Con el primer semáforo elegimos cual de todos los procesos va a ser el proceso candidato (el primero que llegue al semáforo) y cuales los votantes. El proceso candidato debe enviar la señal SIGUSR2 a los votantes para que ejecuten sus votos para luego recoger esos votos e imprimirlos por pantalla. Finalmente, envía la señal SIGUSR1 para repetir el proceso en una nueva ronda.

Por otra parte, los procesos votantes utilizan otro semáforo para escribir su voto en el fichero en orden y que no ocurran errores de pérdidas de datos. Para generar aleatoriamente los votos utilizamos "srand(getpid() + time(NULL))" y en función de si es par o impar escribimos 'N' o 'Y'.

```
int chooseCandidato(int fd, sem_t *sem1, sem_t *sem2, sem_t *sem3, int nProc,
sigset_t mask, int *ronda){
    char answer;
    int i;
    int yes = 0, no = 0;
    int sig, val, *pids = NULL;
    char *votes = NULL;

    // LEEMOS LOS PIDS DEL FICHERO
```

```
    pids = readpids(fd, &nProc);
    if(pids == NULL) {
        perror("readpids");
        return 1;
    }
    // SEMÁFORO = 1 -> CANDIDATO
    if(sem_trywait(sem1) == 0) {

        if(sem_getvalue(sem3, &val) == -1) {
            free(pids);
            return 1;
        }
        while (val>0) {
            if(sem_getvalue(sem3, &val) == -1) {
                free(pids);
                return 1;
            }
        }

        for (int i = 0; i < nProc+1; i++) {
            newVotante = 0;
            if(pids[i] != getpid()) {
                if(kill(pids[i], SIGUSR2) == -1) {
                    perror("kill");
                    free(pids);
                    return 1;
                }
            }
        }

        fd = open(FICHERO, O_CREAT | O_RDONLY, S_IRUSR | S_IWUSR);
        if(fd == -1) {
            perror("open");
            free(pids);
            return 1;
        }
```

```
        int escrito = 0;

        while(escrito==0){
            escrito = 1;
            votes = readvotes(fd, &nProc, ronda);
            if(votes == NULL) {
                perror("readvotes");
                free(pids);
                close(fd);
                return 1;
            }
            for (i =0; i<nProc; i++){
                if (votes[i] != 'Y' && votes[i] != 'N'){
                    escrito = 0;
                    break;
                }
            }
            usleep(1000);
        }

        close(fd);

        printf("Candidate %d => [", getpid());
        for (int i = 0; i < nProc; i++) {
            printf(" %c ", votes[i]);
        }
        printf("] => ");

        for (i=0; i<nProc; i++) {
            if (votes[i] == 'Y') {
                yes++;
            } else {
                no++;
            }
        }

        if(yes > no) {
```

```
            printf("Accepted\n");
        } else {
            printf("Rejected\n");
        }
        fflush(stdout);
        usleep(250000);

        newVotante = 1;

        if(sem_getvalue(sem2, &val) == -1) {
            free(pids);
            free(votes);
            close(fd);
            return 1;
        }
        while (val != 0) {
            sem_wait(sem2);
            if(sem_getvalue(sem2, &val) == -1) {
                free(pids);
                free(votes);
                close(fd);
                return 1;
            }
        }
        sem_post(sem2);

        if(sem_getvalue(sem3, &val) == -1) {
            free(pids);
            free(votes);
            close(fd);
            return 1;
        }
        while (val != nProc) {
            sem_post(sem3);
            if(sem_getvalue(sem3, &val) == -1) {
                free(pids);
                free(votes);
```

```
                close(fd);
                return 1;
            }
        }

        // Enviar SIGUSR1 a cada proceso votante no candidato
        for (int i = 0; i < nProc + 1; i++) {
            if (kill(pids[i], SIGUSR1) == -1) {
                free(pids);
                free(votes);
                close(fd);
                return 1;
            }
        }

        free(pids);
        free(votes);
        sem_post(sem1);
        return 0;
    /// SEMAFORO = 0 -> VOTANTES
    } else {

        sem_wait(sem3);
        sigaddset(&mask, SIGUSR2);
        if (sigwait(&mask, &sig) != 0) {
            perror("sigwait");
            free(pids);
            close(fd);
            return 1;
        }
        sem_wait(sem2);

        srand(getpid()+time(NULL));
        if(rand() % 2==0) {
            answer = 'N';
        } else {
            answer = 'Y';
```

```
        }

        fd = open(FICHERO, O_WRONLY | O_APPEND , S_IRUSR | S_IWUSR);  // Abrir
archivo para añadir votos en modo sobreescritura
        if (fd == -1) {
            perror("open");
            free(pids);
            close(fd);
            return EXIT_FAILURE;
        }

        dprintf(fd, "%d vota %c\n", getpid(), answer);
        close(fd);

        sem_post(sem2);

        sigdelset(&mask, SIGUSR2);
        if (sigwait(&mask, &sig) != 0) {
            perror("sigwait");
            free(pids);
            close(fd);
            exit(EXIT_FAILURE);
        }

        free(pids);
        free(votes);
        return 0;
    }

}
```

VOTANTE

Primero, en esta función, debemos inicializar la máscara con las señales SIGUSR1 y SIGUSR2 para así al recibir la señal SIGUSR1 creamos un bucle para que se repitan las rondas hasta que se acabe el tiempo.

```
int votante(int fd, sem_t *sem1, sem_t *sem2, sem_t *sem3, int nProc, sigset_t
```

```
mask) {
    struct sigaction act_int, act_usr1, act_usr2, act_term, act_alarm;
    int sig;
    sigset_t old_mask;
    int ronda = 0;

    // Inicializar acciones para las señales
    act_int.sa_handler = handle_sigint;
    sigemptyset(&(act_int.sa_mask));
    act_int.sa_flags = 0;

    act_usr1.sa_handler = handle_sigusr1;
    sigemptyset(&(act_usr1.sa_mask));
    act_usr1.sa_flags = 0;

    act_usr2.sa_handler = handle_sigusr2;
    sigemptyset(&(act_usr2.sa_mask));
    act_usr2.sa_flags = 0;

    act_term.sa_handler = handle_sigterm;
    sigemptyset(&(act_term.sa_mask));
    act_term.sa_flags = 0;

    act_alarm.sa_handler = handle_sigalarm;
    sigemptyset(&(act_alarm.sa_mask));
    act_alarm.sa_flags = 0;

    // Configurar sigactions
    if (sigaction(SIGINT, &act_int, NULL) < 0) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }

    if (sigaction(SIGTERM, &act_term, NULL) < 0) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }
```

```
    if (sigaction(SIGALRM, &act_alarm, NULL) < 0) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }

    if (sigaction(SIGUSR1, &act_usr1, NULL) < 0) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }

    if (sigaction(SIGUSR2, &act_usr2, NULL) < 0) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }

    // Bloquear señales durante las oper aciones críticas
    sigemptyset(&mask);
    sigaddset(&mask, SIGUSR1);
    sigaddset(&mask, SIGUSR2);
    sigprocmask(SIG_BLOCK, &mask, &old_mask);

    // Esperar señales
    while (1) {
        sigwait(&mask, &sig);

        if (sig == SIGUSR1) {
            // Repetir proceso de votación si hay un nuevo votante
            while (newVotante) {
                sleep(1);
                ronda ++;
                if(chooseCandidato(fd, sem1, sem2, sem3, nProc, mask, &ronda)
== 1) {
                    exit(EXIT_FAILURE);
                }
            }
        }
```

```
        // Uso de sigsuspend para esperar señales sin consumir CPU
        sigsuspend(&old_mask);    // Esperar señal mientras bloqueamos las
señales en old_mask
    }

    exit(EXIT_SUCCESS);
}
```

EJECUCIÓN

Para ejecutar el programa desde la terminal, tenemos que ejecutar ./voting seguido de dos argumentos que serán el número de procesos votante y el tiempo en segundos hasta que la alarma pare el programa.

Además, si los argumentos son incorrectos, hemos establecido que se vuelvan a pedir los parámetros hasta que sean correctos. Hemos decidido que el número total de votantes esté entre 1 y 29 por el máximo del array de procesos. Además, el tiempo debe ser mayor que cero.

Aquí podemos ver lo que ocurre si se introducen parámetros incorrectos:

```
sara@Sara:/mnt/c/users/saras/onedrive/escritorio/final/SOPER/practica2$ ./vo
ting 30 2
Error al introducir los parámetros, vuelve a intentarlo.
El número de votantes debe encontrar entre 1 y 29
2 3
Candidate 39337 => [ Y  N ] => Rejected
Candidate 39336 => [ N  N ] => Rejected
Finishing by alarm
```

```
sara@Sara:/mnt/c/users/saras/onedrive/escritorio/final/SOPER/practica2$ ./vo
ting 4
Error al introducir los parámetros, vuelve a intentarlo.
El número de votantes debe encontrar entre 1 y 29
Los segundos deben ser mayor que 0
5 6
Candidate 39492 => [ Y  N  N  N  Y ] => Rejected
Candidate 39490 => [ N  N  N  Y  N ] => Rejected
^CFinishing by signal
```

EJEMPLO

A continuación, se muestra un ejemplo de cómo funciona el programa con 5 procesos y un tiempo máximo de 3 segundos.

./voting 5 3

El fichero que se creará con el nombre PIDS será:

PIDS.txt

```
5
10762 10763 10764 10765 10766 10767
10762 vota Y
10763 vota Y
10764 vota Y
10766 vota Y
10767 vota Y
10762 vota Y
10763 vota Y
10764 vota N
10766 vota Y
10767 vota N
```

Y por tanto la salida por terminal es:

```
xiomara@Xiomara-lap:/mnt/c/users/lizbe/DobleG/SOPER/practica2$ ./voting 5 3
Candidate 10765 => [ Y  Y  Y  Y  Y ] => Accepted
Candidate 10765 => [ Y  Y  N  Y  N ] => Accepted
Finishing by alarm
```

APARTADO e)

I) Hay varios momentos donde se pueden producir problemas de concurrencia o pérdida de información, El primer punto es cuando elegimos al candidato ya que varios procesos reciben la misma señal y pueden intentar acceder al código del candidato. Otro punto crítico es cuando se escriben los votos en el fichero porque todos los procesos votantes que en la mayoría de los casos son más de uno, intentan escribir en el mismo fichero y si todos acceden a la vez, se sobrescribirán entre ellos. Por último, al liberar los recursos también podría haber problemas a la hora de recibir las señales.

II) En este tipo de problemas no es muy útil utilizar únicamente señales pues no controlamos los tiempos de estas. Sin embargo, sí que es útil utilizar semáforos para controlar que dos procesos no intentan acceder a la vez a un mismo procedimiento como elegir candidato o escribir en el fichero.