Teoría

Ejercicio 1: Indique una ventaja del uso de bibliotecas dinámicas.

Ejercicio 2: ¿En qué nivel de planificación (corto plazo, medio plazo o largo plazo) se gestiona la expulsión de procesos al área de swap?

Ejercicio 3: Enumere las dos razones para colocar un elemento de información **fuera** del BCP (bloque de control de procesos) de un determinado proceso.

Ejercicio 4: Dibuje el diagrama de estados del ciclo básico de un proceso, para el caso de un único procesador, incluyendo exclusivamente los estados que afectan a la planificación a corto plazo.

Ejercicio 5: Indique un ejemplo de caso en que se produce un cambio de contexto voluntario.

Ejercicio 6: Si un programa que utiliza hilos no realiza llamadas al sistema bloqueantes, ¿qué es mejor usar hilos de usuario (ULT) o hilos de kernel (KLT)?

Ejercicio 7: ¿Cómo se produce una llamada al sistema operativo? Explique el proceso.

Ejercicio 7. Responder brevemente a las siguientes preguntas:

A- ¿Qué es un trap?
B- ¿Qué es el BCP? ¿Qué información almacena?
C- ¿Cuándo se produce un cambio de contexto involuntario de un proceso?
D.- Indica y explica 3 medidas de valoración de la planificación de procesos.

Propuestos del tema de procesos

Ejercicio 1

Se desea desarrollar un servidor Web de alto rendimiento. Para su desarrollo ya se dispone de una biblioteca cuyo archivo de cabecera es el siguiente:

```
#ifndef WEBUTIL_H
#define WEBUTIL_H

struct peticion_web {
  /* Datos de la petición */
};
typedef struct peticion_web peticion_web_t;

void recibir_peticion(peticion_web_t * pet);
void enviar_fichero(peticion_web_t *);

#endif
```

La función `recibir_peticion()` es una función bloqueante que devuelve el control al llamante cuando se recibe una petición, cuyos datos deja en la estructura apuntada por `pet`.

La función `enviar_fichero()` determina la respuesta que hay que enviar al cliente y realiza dicho envío. Dicha función es también bloqueante.

Diseñe e implemente una solución basada en procesos en la que cada vez que se reciba una petición se creará un proceso para procesar dicha petición. De esta manera el proceso principal estará siempre atendiendo la llegada de nuevas peticiones, excepto el tiempo de creación de nuevos procesos auxiliares.

Solución

Ejercicio propuesto.

PARCIAL OCTUBRE 2009

Ejercicio 8 [3 puntos]: Dado el siguiente código, responder a las preguntas que aparecen a continuación:

```
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#define MAX 1
int espera = 10;
void controlador ();
int main (int contargs, char *args[]){
    pid_t pid;
    int i=0;
    signal (SIGCHLD, controlador);

    for (i=0;i<MAX;i++){
       pid = fork ();
       if (pid == 0){
         while(1){
           sleep (1);
           printf("Soy %d \n", i);
         }
         exit(0);
       }
     }

     sleep (espera);
     signal (SIGCHLD, SIG_IGN);
     kill (pid, SIGINT);
     exit (0);
 }
 void controlador (){
     int id, est;
     id = wait (&est);
     exit (0);
 }
```

a) Describa el funcionamiento del programa
b) ¿Qué pasaría si MAX tuviera un valor de 10?
c) Modifique el programa para que funcione para cualquier valor de MAX.

a) El proceso padre crea un hijo, y después de 10 segundos le manda una señal para que termine su ejecución. Si se tiene en cuenta `signal (SIGCHLD, SIG_IGN);` el programa padre termina. En caso contrario, cuando el hijo muera, se ejecutará el código de la función controlador.

b) Si MAX es igual a 10, se crearán 10 procesos hijos de los que solo 1 terminará.

c) Se proponen 2 soluciones:

```
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#define MAX 100
int espera = 10;
pid_t pid[MAX];

void controlador ();
int main (int contargs, char *args[]){
    int i=0;
    signal (SIGCHLD, controlador);
    for (i=0;i<MAX;i++){
       pid[i] = fork ();
       if (pid[i] == 0){
         while(1){
           sleep (1);
           printf("Soy %d \n", i);
         }
         exit(0);
       }
     }
     sleep (espera);
     signal (SIGCHLD, SIG_IGN);
     for (i=0;i<MAX;i++){
        kill (pid[i], SIGINT);
     }
     exit (0);
 }
 void controlador (){
     int id, est, i;
     for (i=0;i<MAX;i++){
       id = wait (&est);
     }
     exit (0);
 }
```

```
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#define MAX 100
int espera = 10;
void controlador ();

int main (int contargs, char *args[]){
    pid_t pid;
    int i=0;
    signal (SIGCHLD, controlador);
    for (i=0;i<MAX;i++){
       pid = fork ();
       if (pid == 0){
         while(1){
           sleep (1);
           printf("Soy %d \n", i);
         }
         exit(0);
       }
       sleep (espera);
       signal (SIGCHLD, SIG_IGN);
       kill (pid, SIGINT);
     }
     exit (0);
 }
 void controlador (){
     int id, est;
     id = wait (&est);
     exit (0);
 }
```

PARCIAL OCTUBRE 2009

Ejercicio 9 [3 puntos]: En un determinado sistema operativo los procesos se ejecutan con planificación apropiativa y política de planificación cíclica (round-robin).

En la siguiente tabla se especifica para cada proceso, su tiempo de llegada y el tiempo que necesitan para ejecutarse. Todos los procesos realizan exclusivamente tareas de cálculo.

Proceso	Tiempo de llegada	Tiempo de ejecución
P1	0	500
P2	100	300
P3	300	400
P4	600	1000
P5	700	600

Se desea evaluar las diferencias que se producirán al variar la longitud de la rodaja de tiempo, considerándose valores de 200 y 500 milisegundos.

Para las dos posibilidades, se pide:

1. Determine el tiempo de finalización de cada proceso.
2. Determine el tiempo que cada proceso ha estado en el sistema (tiempo de retorno).
3. Determine el tiempo de servicio y el tiempo de espera de cada proceso.
4. Determine el tiempo de retorno normalizado
5. Determine el tiempo medio de espera.
6. Determine el tiempo medio de retorno normalizado.

¿Puede concluir algo de los resultados?

T (rodaja 200)	CPU	COLA
0	P1<500>	
100	P1<400>	P2<300>
200	P2<300>	P1<300>
300	P2<200>	P1<300>, P3<400>
400	P1<300>	P3<400>, P2<100>
600	P3<400>	P2<100>, P4<1000>, P1<100>
800	P2<100>	P4<1000>, P1<100>, P5<600>, P3<200>
900 – Fin P2	P4<1000>	P1<100>, P5<600>, P3<200>
1100	P1<100>	P5<600>, P3<200>, P4<800>
1200 – Fin P1	P5<600>	P3<200>, P4<800>
1400	P3<200>	P4<800>, P5<400>
1600 – Fin P3	P4<800>	P5<400>
1800	P5<400>	P4<600>
2000	P4<600>	P5<200>
2200	P5<200>	P4<400>
2400 – Fin P5	P4<400>	
2800 – Fin P4		

Proceso	Ti	Tf	Tq	Ts	Te	Tq norm
P1	0	1200	1200	500	700	1200/500 = 2,4
P2	100	900	800	300	500	800/300 = 2,67
P3	300	1600	1300	400	900	1300/400 = 3,25
P4	600	2800	2200	1000	1200	2200/ 1000 = 2,2
P5	700	2400	1700	600	1100	1700/600 = 2,83
Promedio					880	2,67

T (rodaja 500)	CPU	COLA
0	P1<500>	
100	P1<400>	P2<300>
300	P1<300>	P2<300>, P3<400>
500 – Fin P1	P2<300>	P3<400>
600	P2<200>	P3<400>, P4<1000>
700	P2<100>	P3<400>, P4<1000>, P5<600>
800 – Fin P2	P3<400>	P4<1000>, P5<600>
1200 – Fin P3	P4<1000>	P5<600>
1700	P5<600>	P4<500>
2200	P4<500>	P5<100>
2700 – Fin P4	P5<100>	
2800 – Fin P5		

Proceso	Ti	Tf	Tq	Ts	Te	Tq norm
P1	0	500	500	500	0	500/500 = 1
P2	100	800	700	300	400	700/300 = 2,33
P3	300	1200	900	400	500	900/400 = 2,25
P4	600	2700	2100	1000	1100	2100/1000 = 2,1
P5	700	2800	2100	600	1500	2100/600 = 3,5
Promedio					700	2,23

Ejercicio 2. Dado el siguiente programa, y considerando que cada vez que un proceso tiene el procesador ejecuta la instrucción de suma *(var++)* 100.000.000 de veces por segundo, completar la tabla proporcionada en los siguientes casos:

a) Planificación con prioridades expulsiva.

b) Round Robin con rodaja de tiempo de 2 seg, expulsiva.

```
int var=0; //variable global
int fin=0; //variable global
void terminar(void) {
        // cuando se termina un proceso se imprime el valor de 'var'
        printf("Proceso %d: var=%d\n", getpid(), var);
        fin = 1;
}
main() {
        int pid,i;
signal(SIGALRM, terminar); /* se establece la acción a ejecutar cuando se reciba una alarma.
Esta acción es la función terminar() */
        for(i=0; i<3; i++) {
                sleep(1);
               pid=fork();
               switch(pid) {
                case -1:
                        perror("Error en la creación de procesos");
                        exit(-1);
                case 0:
                       alarm(5);  /* Se ejecuta siempre de inmediato al crear el hijo porque
                                cada proceso que se crea tiene inicialmente la misma prioridad
                                que el padre (prioridad = P) */
                        nice(i%2 + 1); /*Disminuye la prioridad del proceso tantas unidades
                                como se indique en el parámetro*/
                        while(!fin)
                             var++; /* Se ejecuta la iteración hasta que fin cambia de valor, es
                          decir, cuando se recibe la alarma y se trata en la función 'terminar' */
                 exit(0);

                } //fin switch
        } //fin for
}  fin main
```

TABLA A COMPLETAR

PROCESO	**INSTANTE DE LLEGADA**	**PRIORIDAD**	**INSTANTE FINAL DE EJECUCIÓN**	**VALOR DE VAR IMPRESO - Apartado a) -**	**VALOR DE VAR IMPRESO - Apartado b) -**
Padre	0	P	3	no imprime var	no imprime var
Hijo 0	1	P - 1	6		
Hijo 1				0	
Hijo 2	3				

Solución:

PROCESO	**INSTANTE DE LLEGADA**	**PRIORIDAD**	**INSTANTE FINAL DE EJECUCIÓN**	**VALOR DE VAR IMPRESO - Apartado a) -**	**VALOR DE VAR IMPRESO -Apartado b) -**
Padre	0	P	3	no imprime var	no imprime var
Hijo 0	1	P - 1	6	500.000.000	300.000.000
Hijo 1	2	P - 2	7	0	0
Hijo 2	3	P - 1	8	200.000.000	400.000.000

En el caso B también se puede dar como admisible que el valor impreso de *var* por el hijo 0 sea 400.000.000 y el del hijo 2 sea 300.000.000.

Ejercicio . Dado el programa que se muestra a continuación:

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

main () {
  int pid,i;

  for (i=0; i<3; i++){
    printf("INICIO\n");
    pid=fork();
    if (pid == 0){
       sleep (1);
       printf("UNO\n");
    }
    else {
       sleep (2);
       printf("DOS\n");
    }
  }
}
```

A- Indicar cuantas veces aparece la palabra UNO en pantalla y cuantas veces aparece la palabra DOS y en que momentos de la ejecución, tomando como 0 el momento en el que se escribe INICIO

B- Indicar la jerarquía de procesos creados utilizando la notación padre, hijo, nieto, bisnieto y mostrando claramente las relaciones jerárquicas y el orden de creación de los procesos dentro de su mismo nivel.