Universidad Nacional de La Matanza
Florencio Varela 1903 - San Justo - Buenos Aires - Argentina

Ingeniería en Informática

Sistemas Operativos

Trabajo Investigación

Planificación de Procesos con FCFS-SJF y Estadísticas del CPU

Equipo Docente	Nicanor Casas
	Graciela De Luca
	Waldo Valiente
	Gerardo Puyo
	Sergio Martín
	Federico Díaz
	Sebastian Deuteris

GRUPO: 16
Días de Cursada: ***Miércoles/Viernes***

	Alumnos	
Apellido	**Nombre**	**DNI**
Plastina	Damian	33.779.183
Sandler	Alejandro	35.169.977
Lagomarsino	Ezequiel	34.258.658
Espinosa	Pablo	34.575.756

Acreditación:

Instancia	Fecha	Calificación
PRE-ENTREGA	/ /2012	
ENTREGA	/ /2012	
FINAL	/ /2012	

Índice

Objetivo 3

Alcance 3

Algoritmo de planificación First-Come, First-Served (FCFS) 4

Introducción 4

Resumen de modificaciones y agregados 4

Variables 4

Funciones 5

Prueba del algoritmo 8

Conclusión 10

Algoritmo de Planificación Shortest Job First (SJF) 11

Introducción 11

Resumen de modificaciones y agregados 11

Variables 11

Funciones 12

Prueba del algoritmo 16

Conclusión 18

Propuesta de mejoras 18

Estadísticas de los procesos y de la CPU 20

Introducción 20

Resumen de modificaciones y agregados 20

Variables 20

Funciones 21

Modo de ejecución 21

Objetivo

El objetivo de este trabajo práctico consiste en implementar dos nuevos planificadores de procesos denominados First-Come, First Served (FCFS) y Shortest Job First (SJF). Estos se desarrollarán de forma tal que coexistan con los ya implementados y puedan ser intercambiados en tiempo de ejecución. Además se proporcionara información estadística acerca del uso de la CPU.

Alcance

- Implementar dos nuevos planificadores (FCFS y SJF).
- Proveer los mecanismos que permitan cambiar de planificador en tiempo de ejecución.
- Asegurarse que las modificaciones no afecten la ejecución de otros planificadores.
- Proporcionar información estadística de uso de la CPU.

Algoritmo de planificación First-Come, First-Served (FCFS)

Introducción

El algoritmo de planificación *FCFS* le asigna al procesador al primero que esté listo para el uso del mismo. El seleccionado se ejecutará hasta que termine por completo, o realice una solicitud de Entrada/Salida y los demás quedarán esperando.

Resumen de modificaciones y agregados

Variables

Archivo: /include/kernel/system.h

Variable: Se agrega la variable *unsigned long ulContadorListo*

Objetivo: Esta se utiliza para contabilizar los cambios de estado de los procesos a Listo. El número almacenado en la variable se le otorgará a cada proceso para que de esta forma se establezca el orden de ejecución.

Archivo: /include/kernel/sched.h

Variable: Se declara el siguiente *#define FCFS 0*

Objetivo: El motivo es agregar la identificación del planificador correspondiente

Archivo: /include/kernel/pcb.h

Variable: Se agrega a la estructura *_stuPCB* la variable *unsigned long ulTiempoListo*

Objetivo: El propósito es almacenar en el PCB del proceso el número correspondiente al orden de llegada a la cola de listos. Que luego se utiliza por el planificador para establecer quien pasará a ejecución.

```
typedef struct _stuPCB_ {
…
unsigned long ulTiempoListo; /*!< tiempo en que el proceso pasa
al estado de listo */
…
}
```

Funciones

Archivo: múltiples **[Tabla 1]**

Funciones: múltiples **[Tabla 1]**

Objetivo: Para implementar este planificador es necesario saber en qué momento los procesos pasan a listo. Por ello se modificaron todas las funciones en donde esto sucede, agregándoles el siguiente código, que permite guardar el instante en que esto ocurre.

```
pstuPCB[indice].iEstado  = PROC_LISTO;
//guardo el momento en que  el proceso pasa a listo
pstuPCB[indice].ulTiempoListo=ulContadorListo;
ulContadorListo++;  //se incrementa el contador
```

Archivo	Función	Líneas
kernel/gdt.c	iFnCrearPCB()	518 – 519 – 520
	iFnEliminarProceso()	2014 – 2015 – 2016
kernel/semaforo.c	iFnSemPost()	227 – 228 – 229
	iFnSemClose()	103 – 104 – 105
kernel/syscall.c	lFnSysExit()	118 – 119 – 120
	lFnSysKill()	751 – 752 – 753
	lFnSysIdle()	1711 – 1712 – 1713
kernel/mem/sodheap_kernel.c	vFnKFree()	203 – 204 – 205
kernel/drivers/teclado.c	vFnManejadorTecladoShell()	117 – 118 – 119
kernel/planif/bts.c	vFnPlanificadorBTS()	107 – 108
kernel/planif/rr.c	vFnPlanificadorRR()	103 – 104
kernel/planif/rrprio.c	vFnKFree()	145 – 146
kernel/system.c	vFnAjustarReloj()	1075-1076

[Tabla 1] Archivos y funciones afectadas por el cambio de estado

Archivo: usr/bin/chppalg.c

Función: iFnValidaParametros()

Objetivo: Esta modificación es necesaria para que el comando reconozca FCFS como opción para el usuario.

```
int iFnValidaParametros(char *cpParametro)
{
 …
        if(iFnCompararCadenas(cpParametro, "FCFS") != 1)
            {
                  …
            }
….
return 1;
}
```

Archivo: usr/bin/chppalg.c

Función: main()

Objetivo: El siguiente agregado es para que comando pueda reconocer FCFS como planificador utilizado en el momento de ser utilizado.

```
  {
   ….
   if(iFnCompararCadenas(argv[1], "-a") == 1)
      {
      …
         switch (iPlanificadorActivo)
         {
            case (FCFS):
               vFnImprimir("First Come First Serve (FCFS)");
               break;
       }
   …..
   …
```

Objetivo: Se inserta el código para que aparezca FCFS como opción cuando el comando liste los planificadores disponibles.

```
if(iFnCompararCadenas(argv[1], "-o") == 1)
   {
      vFnImprimir("\n OPCION   DESCRIPCION");
      vFnImprimir("\n ------   -----------");
      vFnImprimir("\n FCFS    :Algoritmo que asigna el
procesador al ");
      vFnImprimir("primer proceso que haya");
….
}
…
..
```

Objetivo*: S*e agrega el código para que en el momento que se seleccione FCFS como planificador a utilizar, se valide no se esté utilizando en ese momento.

```
int iPlanificadorActivo;
   iPlanificadorActivo = sched_getscheduler();
   switch (iPlanificadorActivo)
   {
     case (FCFS):
        if(iFnCompararCadenas(argv[1], "FCFS") == 1)
        {
         vFnImprimir("\n  La  opcion  elegida  ya  se  encuentra
activa en sistema.");
         return 0;
        }
        break;
……
   }

…..
..
```

Objetivo: El agregado permite establecer a FCFS como el actual modo planificación.

```
int iRespuestaCambioPlanificador;
   if(iFnCompararCadenas(argv[1], "FCFS") == 1)
   {
      iRespuestaCambioPlanificador = sched_setscheduler(FCFS);
   }
….
..
}
```

Archivo: /kernel/system.c

Función: vFnHandlerTimer ()

Objetivo: Si el planificador no es por tiempo y el proceso nulo llega a ejecutar, este toma el control del procesador y el sistema operativo nunca volvería a recuperarlo. Por lo tanto, se implementa este mecanismo para evitar que el proceso nulo ejecute infinitamente.

```
vFnHandlerTimer ()
{
  vFnAjustarReloj ();

  if (siFnEsPlanificadorPorTiempo (siPlanificador))
    {
      vFnEjecutarPlanificadorPorTiempo ();
    }

  else if (lFnSysGetPPid () == 0 ||
           (lFnSysGetPPid  ()  !=  0  &&  pstuPCB[ulProcActual].iEstado
!= 0))
    {
```

```
        if(ulContadorListo>=1000000)
        {
           ulContadorListo=0;
        }
          vFnPlanificador ();
    }
```

Archivo: /kernel/main.c
Función: main()
Objetivo: se inicializa a cero la variable ulContadorListo-

Archivo: /kernel/planif/fcfs.c

Función: vFnPlanificadorFCFS()

Objetivo: Se crea el archivo “fcfs.c” que contiene la función “vFnPlanificadorFCFS” quien es la que se encarga de la planificación, de la selección del próximo proceso a ejecutar. Para esto, busca el proceso con el menor valor en el atributo ulTiempoListo y a continuación es colocado en el procesador.

Retorno: Ninguno.

Prueba del algoritmo

Para probar el algoritmo, se crearon 3 procesos usuarios, donde cada uno hace:

Prog1: 1) Ejecuta.
2) Duerme 10 segundos.
2) Ejecuta 2 segundos.

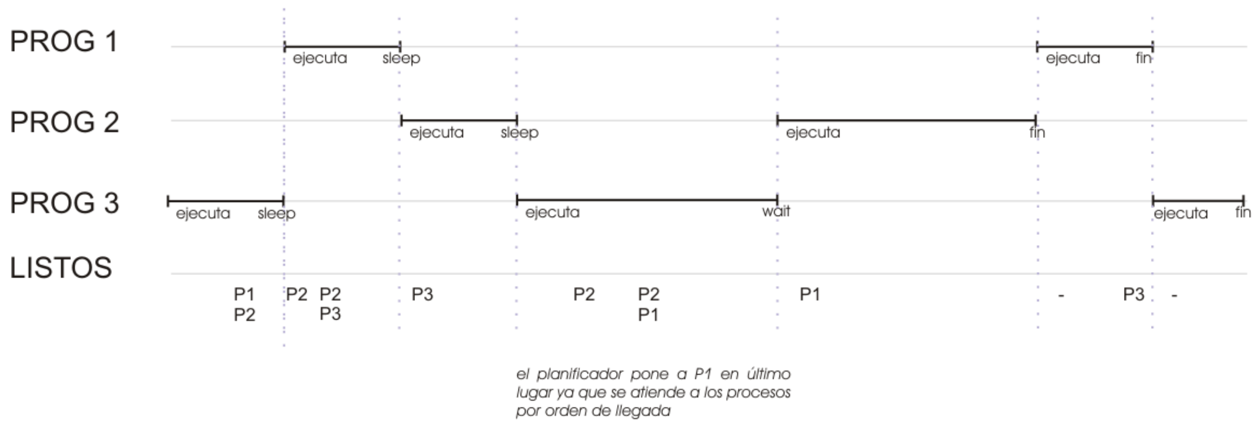
Prog2: 1) Ejecuta
2) Duerme 7 segundos.
3) Ejecuta 13 segundos.

Prog3: 1) Crea Prog1 y Prog2.
2) Duerme
2) Ejecuta 16 segundos.
3) Espera a Prog2 y Prog1.

Aclaración: Los tiempos están dados para un procesador Phenom2 X4 965. La idea es que haya procesos con distintos tiempos de ejecución y de espera.

El procedimiento a realizar es ejecutar Prog3, este llama a Prog1 y Prog2, luego teniendo en cuenta el algoritmo utilizado, en este caso FCFS, se esperará que después de ejecutar Prog3, se ejecute Prog2 y luego se ejecutara Prog1, siguiendo el orden que establece el algoritmo. Teniendo una secuencia como lo muestra la siguiente figura:

Planificación de procesos para FCFS



[Fig. 1]

El resultado esperado es que al ejecutar Prog3, Prog1 y Prog2 están en estado de Listo compitiendo por el procesador, en ese momento el planificador selecciona el que haya estado Listo primero, entonces elige Prog2 y luego prog1, mostrándolo por pantalla como la siguiente figura:

```
Bochs x86 emulator, http://bochs.sourceforge.net/
CSW: PID 0      TAREA=IDLE    BIN       INDICE GDT: 0x8        25/11/12 14:48:30

Bienvenidos al SODIUM...
Cmd>
Cmd>
Cmd>
Cmd>chppalg FCFS
 Cambio exitoso!
Cmd>Prog3
Hola soy PROG3 y voy a llamar a PROG1 y PROG2
Hola soy PROG2
PROG2 - me voy a dormir 55 ...
Hola soy PROG1
PROG1 - me voy a dormir 77 ...
** PROG3 - mientras yo ejecuto, PROG1 y PROG2 se despiertan **
** PROG3 - termine de ejecutar **
PROG2 - me despierto y ejecuto 5999999 ciclos
Chau soy PROG2
PROG1 - me despierto y ejecuto 999999 ciclos
Chau soy PROG1
Chau soy PROG3

Cmd>
| ayuda | ps  |  init | mem | segs  | cls  | F2: Log | Mouse: è/%4C4C+)*4613
IPS: 13.067M    A:  NUM  CAPS  SCRL  UHCI  OHCI
```

[Fig. 2]

Conclusión

El planificador FCFS no es muy eficiente, ya que este genera gran tiempo de espera, y lo que se conoce como efecto convoy, efecto que ocurre al tener varios procesos limitados por E/S y pocos limitados por CPU, donde en determinados momentos el CPU o los dispositivos de E/S quedan inactivos. Pero es un planificador básico que tiene que estar.

Para su implementación se podría haber utilizado una lista, pero eso generaría mucha sobrecarga al sistema operativo sobre todo porque habría que seguir manteniendo su coherencia a medida que se cambia a otros planificadores.

Como posible mejora se le podría atribuir prioridades. Es decir, que el algoritmo seleccione como debe, y además teniendo en cuenta las prioridades de los procesos. De esta forma, el proceso que haya estado primero en estado Listo y tenga a la vez mayor prioridad, será el que pasará a manos del procesador.

Algoritmo de Planificación Shortest Job First (SJF)

Introducción

El algoritmo de planificación Shortest Job First selecciona el próximo proceso a alojar en la CPU en base a la duración del mismo. A cada proceso se le asocia una duración de su próxima ráfaga de ejecución, según la cual se determinará la prioridad del proceso para la utilización de la CPU. Los procesos cuya próxima ráfaga de ejecución sea más corta tendrán mayor prioridad.

Resumen de modificaciones y agregados

Variables

Archivo: /include/kernel/pcb.h

Variable: Se agrega a la estructura *_stuPCB* la variable *unsigned long ulTiempoListo*

Objetivo: El propósito es almacenar en el PCB del proceso información sobre tiempo de inicio y finalización de la ráfaga de CPU, duración de la última ráfaga, el acumulado, la cantidad de ráfagas y el promedio, para luego poder tomar la decisión de cuál será el proceso a ejecutarse según el tiempo promedio de duración de ráfagas, para que de esta forma se pueda implementar el planificador:

```
typedef struct _stuPCB_ {
…
unsigned long sjfInicioRafaga;/*!<Instante en que comienza la rafaga
actual*/
unsigned long sjfFinRafaga;/*!<Instante en que finaliza la rafaga actual*/
unsigned long sjfTiempoUltimaRafaga;/*!<Duración de la última ráfaga de
CPU*/
unsigned long sjfAcumuladoRafagas;/*!<Tiempo Acumulado de todas las ráfagas.
Para el cálculo del promedio*/
int sjfCantRafagas;/*!<Cantidad de ráfagas. Para el cálculo del promedio*/
unsigned long sjfPromedioRafagas;/*!<Tiempo Promedio de todas las ráfagas.
Conservo el promedio para no calcularlo en cada ciclo al recorrer los PCBs*/
int statPosicion; /*!<Ubicacion en el vector de estadisticas*/

…
```

Archivo: /include/kernel/sched.h

Variable: Se declara el siguiente *#define SJF 5*

Objetivo: El motivo es agregar la identificación del planificador correspondiente.

```
#define SJF 5 /*!< Valor del planificador SJF*/
```

Archivo: /include/kernel/gdt.h

Objetivo: Se crea una estructura de datos, que será un vector, que se utilizará para almacenar las estadísticas para implementar el planificador.

```
typedef struct _pStat_

{
char libre;
char nombre[25];
unsigned long tiempoRafagaPromedio;
unsigned long tiempoRafagaAcumulado;
int cantidadRafagas;
} pStat;

pStat *procStats; /*!< Vector de estructuras de estadisticas para cada proceso*/
```

Archivo: /include/kernel/definiciones.h
Objetivo: Se agrega un define que hace referencia al tamaño máximo del vector de estadísticas.

```
#define CANTMAXSTATPROCS 100
```

Archivo: /kernel/main.c
Variable: Se declara la variable uiIndiceVectorStat como unsigned int.
Objetivo: Esta se utilizará como índice en el vector de estadísticas.

Funciones

Archivo: usr/bin/chppalg.C
Función: iFnValidaParametros()
Objetivo: Se realiza el cambio con el motivo de que se reconozca SJF como parámetro correcto al llamar el comando “chppalg”. Para esto se agrega un if para comparar el parámetro con “SJF” para validar que sea correcto.

```
....
 {
    if(iFnCompararCadenas(cpParametro, "SJF") != 1)
    {
       return -1;
    }
...
.
```

Archivo: usr/bin/chppalg.C
Función: Main()
Objetivo: Aquí se agregan distintas cosas:
Se agrega el código necesario para que reconozca a SJF como planificador activo, mediante el agregado de un case en el switch.

```
case (SJF):
vFnImprimir("Shortest Job First");
break;
```

Se realiza al agregado de código necesario para que cuando se quiera enlistar los planificadores disponibles a utilizar se muestre SJF. Para esto se agrega un vFnImprimir.

```
vFnImprimir("\n SJF     :Asigna los procesos en base a estadisticas.");
```

El último cambio es que permite que se pueda elegir SJF como planificador a utilizar y valida que no se esté utilizando en ese momento. Esto se realiza mediante el agregado de un if que compara el parámetro de la función con "SJF"

```
if(iFnCompararCadenas(argv[1], "SJF") == 1)
   {
      iRespuestaCambioPlanificador = sched_setscheduler(SJF);
   }
```

Archivo: /kernel/syscall.c
Función: lFnSysExecve()
Objetivo: Se reemplaza las estadísticas cuando se ejecuta dicha función, al crear un proceso nuevo, ya que si no tendría las del shell.

```
  statPosicion = iFnBuscarStatProceso(strNombreArchivoAux);
  pstuPCB[iFnBuscaPosicionProc(pstuPCB[ulProcActual].ulId)].sjfTiempoUltimaRafaga
= procStats[statPosicion].tiempoRafagaPromedio;
  pstuPCB[iFnBuscaPosicionProc(pstuPCB[ulProcActual].ulId)].sjfAcumuladoRafagas =
procStats[statPosicion].tiempoRafagaAcumulado;
  pstuPCB[iFnBuscaPosicionProc(pstuPCB[ulProcActual].ulId)].sjfCantRafagas =
procStats[statPosicion].cantidadRafagas;
  pstuPCB[iFnBuscaPosicionProc(pstuPCB[ulProcActual].ulId)].sjfPromedioRafagas =
procStats[statPosicion].tiempoRafagaPromedio;
```

Archivo: /kernel/main.c
Funcion: main()
Objetivo: Se agrega en la función el código que se encarga de inicializar el vector de estadísticas.

```
vFnImprimir ("\nInicializando Vector de estadisticas...");
procStats=((pStat*)pvFnReservarPartMemoriaAlta(sizeof(pStat)*CANTMAXSTATPROCS));
for(uiIndiceVectorStat=0;uiIndiceVectorStat<CANTMAXSTATPROCS;uiIndiceVectorStat++)
  {
    procStats[uiIndiceVectorStat].libre = 'S';
  }
```

Archivo: /kernel/system.c
Función: vFnPlanificador()
Objetivo: Se realiza el agregado para que se llame a la función correspondiente a al algoritmo SJF. Esto se hace mediante de un case en el switch.

```
case SJF:
      vFnPlanificadorSJF ();
```

Archivo: /include/kernel/gdt.h
Objetivo: Se agrega el encabezado de la función iFnBuscarStatProceso, que recibe como parámetro el nombre del proceso, y devuelve su posición dentro del vector de estadísticas.

```
int iFnBuscarStatProceso(char [25]);
```

Archivo: /include/kernel/gdt.h
Objetivo: Se modifica el encabezado de la función iFnCrearPCB agregándole como parámetro un entero que representa la posición del proceso dentro del vector de estadísticas.

```
int iFnCrearPCB(int,void*,char*,unsigned int,unsigned int,unsigned
int,unsigned int,unsigned int,unsigned int,unsigned int,unsigned int,
unsigned int, unsigned int,unsigned int,int);
```

Archivo: /kernel/gdt.c
Función: iFnBuscarStatProceso()
Objetivo: Busca el proceso en el vector de estadísticas. Si no lo encuentra crea la entrada.
Retorno: Devuelve la posición en el vector de estadísticas o -1 si no hay posiciones libres

```
int iFnBuscarStatProceso(char nombre[25])
{
  int iPosicion = 0;
  int i = 0;
  int match = 1;

  //Busco el proceso recibido
  while (procStats[iPosicion].libre == 'N' && iPosicion <
CANTMAXSTATPROCS)
  {
    i = 0;
    match = 1;
      while ( match == 1 && ( procStats[iPosicion].nombre[i] != '\0'
|| nombre[i] != '\0' ) )
      {
        if(procStats[iPosicion].nombre[i] != nombre[i])
        {
          match = 0;
        }
        i++;
      }
      if ( match == 1)
      {
        vFnImprimir("P: %d\n",iPosicion);
        return iPosicion;
      }
    iPosicion++;
  }

  //El proceso no existe.
  if( procStats[iPosicion].libre == 'S' )
  {
    //Si hay espacio libre lo creo.
    vFnImprimir("Creacion Stat para %s ",nombre);
    procStats[iPosicion].libre = 'N';
    i = 0;
    while(i < 25)
    {
      procStats[iPosicion].nombre[i] = nombre[i];
      if ( nombre[i] == '\0')
      {
        i = 25;
      }
      else
```

```
        {
          i++;
        }
      }
      procStats[iPosicion].tiempoRafagaPromedio = 10;
      procStats[iPosicion].tiempoRafagaAcumulado = 10;
      procStats[iPosicion].cantidadRafagas = 1;
      vFnImprimir("P: %d\n",iPosicion);
      return iPosicion;
    }
    return -1;
  }
```

Archivo: /kernel/gdt.c
Función: iFnCrearPCB()
Objetivo: Se agrega como parámetro de la función, la variable correspondiente a la posición del proceso dentro del vector de estadísticas (statPosicion).

También se agrega la asignación de los valores predeterminados de las variables correspondientes a las de estadísticas para SJF (sjfInicioRafaga, sjfFinRafaga, sjfTiempoUltimaRafaga, sjfAcumuladoRafagas, sjfCantRafagas, sjfPromedioRafagas, statPosicion).

```
int iFnCrearPCB( int iPosicion,
          void *pEIP,
          char *stNombre,
        ....
        ...
        int statPosicion)
{
.....
.......
  pstuPCB[iPosicion].sjfInicioRafaga =
ulFnSysGetTimeMilisegundos();
  pstuPCB[iPosicion].sjfFinRafaga = NULL;
  pstuPCB[iPosicion].sjfTiempoUltimaRafaga =
procStats[statPosicion].tiempoRafagaPromedio;
  pstuPCB[iPosicion].sjfAcumuladoRafagas =
procStats[statPosicion].tiempoRafagaAcumulado;
  pstuPCB[iPosicion].sjfCantRafagas =
procStats[statPosicion].cantidadRafagas;
  pstuPCB[iPosicion].sjfPromedioRafagas =
procStats[statPosicion].tiempoRafagaPromedio;
....
...

}
```

Archivo: /kernel/gdt.c
Función: iFnInstanciarTrabajadorKernel()
Objetivo: Se agrega la inicialización de la variable statPosicion, con cero. Y luego se le asigna el valor correspondiente por medio de la función iFnBuscarStatProceso. También se agrega el parámetro statPosicion cuando se llama a la función iFnCrearPCB.

```
int statPosicion = 0;
...
```

```
iPosicion = iFnBuscarPCBLibre();
  statPosicion = iFnBuscarStatProceso(stNombre);
  uiIndiceGDT_TSS = uiFnBuscarEntradaGDTLibre();
.....
```

Archivo: /kernel/gdt.c
Función: iFnCrearProceso
Objetivo: Se agrega la inicialización de la variable statPosicion, con cero. Y luego se le asigna el valor correspondiente por medio de la función iFnBuscarStatProceso. También se agrega el parámetro statPosicion cuando se llama a la función iFnCrearPCB

Archivo: /kernel/gdt.c
Función: iFnDuplicarProceso
Objetivo: Se agrega la inicialización de la variable statPosicion, con cero. Y luego se le asigna el valor correspondiente por medio de la función iFnBuscarStatProceso. También se agrega el parámetro statPosicion cuando se llama a la función iFnCrearPCB

Archivo: /kernel/gdt.c
Función: iFnEliminarProceso
Objetivo: Se realiza el cálculo de las estadísticas antes de borrar el proceso, fin de ultima ráfaga, acumulado, cantidad de ráfagas, etc.

```
vFnLog( "\n iFnEliminarProceso(): Eliminando PCB %d (PID %d)",uiProceso,
pstuPCB[ uiProceso ].ulId );
      pstuPCB[ uiProceso ].iEstado = PROC_ELIMINADO;
      pstuPCB[uiProceso].sjfFinRafaga=ulFnSysGetTimeMilisegundos();
      pstuPCB[uiProceso].sjfTiempoUltimaRafaga =
    pstuPCB[uiProceso].sjfFinRafaga -
    pstuPCB[uiProceso].sjfInicioRafaga;
      pstuPCB[uiProceso].sjfAcumuladoRafagas =
    pstuPCB[uiProceso].sjfAcumuladoRafagas +
    pstuPCB[uiProceso].sjfTiempoUltimaRafaga;
      pstuPCB[uiProceso].sjfCantRafagas++;

      if (iFnListaBuscar(
```

Archivo: /kernel/planif/sjf.c
Función: vFnPlanificadorSJF
Objetivo: Se encarga del realizar el algoritmo de selección del próximo proceso a ejecutar en el procesador, siguiendo el orden establecido por SJF teniendo en cuenta el menor de los valores promedio de duración de la ráfaga.
Retorno: Ninguno.

Prueba del algoritmo

Para probar el algoritmo, se crearon 3 procesos usuarios, donde cada uno hace:

Prog1: 1) Ejecuta.
2) Duerme 10 segundos.
2) Ejecuta 2 segundos.

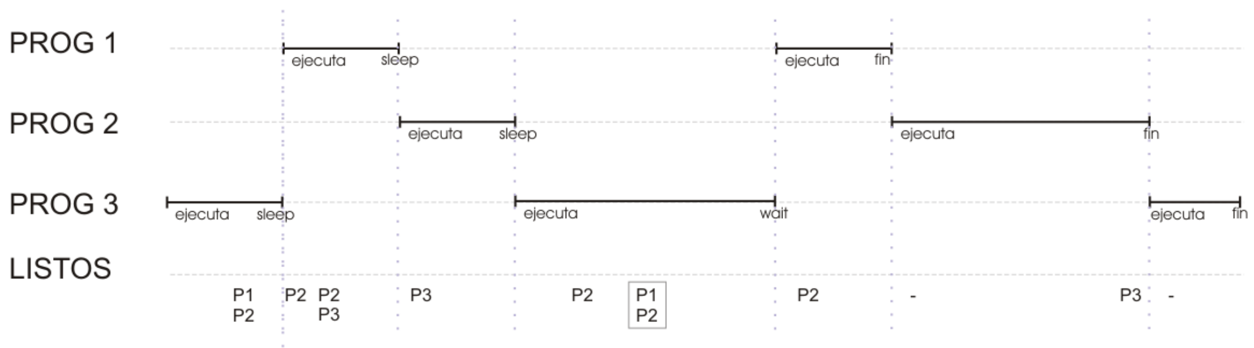
Prog2: 1) Ejecuta
2) Duerme 7 segundos.
3) Ejecuta 13 segundos.

Prog3: 1) Crea Prog1 y Prog2.
2) Duerme
2) Ejecuta 16 segundos.
3) Espera a Prog1 y Prog2.

Aclaración: Los tiempos están dados para un procesador Phenom2 X4 965. La idea es que haya procesos con distintos tiempos de ejecución y de espera.

El procedimiento a realizar es ejecutar Prog3, este llama a Prog1 y Prog2, luego teniendo en cuenta el algoritmo utilizado, en este caso SJF, se esperará que después de ejecutar Prog3, se ejecute Prog1, ya que este tiene mayor tiempo de ejecución, y luego se ejecutará Prog2, siguiendo el orden que establece el algoritmo. Teniendo una secuencia como lo muestra la siguiente figura:

Planificación de procesos para SJF



[Fig. 3]

Hay que tener en cuenta que al ejecutar por primera vez un proceso este tendrá las estadísticas establecidas por default. Teniendo en cuesta esto, al ejecutar por primera vez Prog3 su primer comportamiento será el mismo que el explicado con anterioridad para el algoritmo FCFS. Para las futuras ejecuciones, cada proceso tendrá sus estadísticas basadas en el tiempo que se ejecutó cada uno con anterioridad.

El algoritmo hace el procesamiento de las estadísticas de los procesos solo cuando el algoritmo está activo.

El resultado esperado es que al ejecutar Prog3, se encuentra Prog1 y Prog2 compitiendo por el procesador, en ese momento el algoritmo elige dando lugar honor a su nombre se ejecute Prog1 primero al tener menos tiempo de ejecución que Prog2, mostrándolo como la siguiente figura:

```
Bochs x86 emulator, http://bochs.sourceforge.net/
CSW: PID 0      TAREA=IDLE    BIN       INDICE GDT: 0x8        25/11/12 15:23:43

Cmd>
Cmd>
Cmd>
Cmd>chppalg SJF
 Cambio exitoso!
Cmd>
Cmd>Prog3
Hola soy PROG3 y voy a llamar a PROG1 y PROG2
Hola soy PROG2
PROG2 - me voy a dormir 55 ...
Hola soy PROG1
PROG1 - me voy a dormir 77 ...
** PROG3 - mientras yo ejecuto, PROG1 y PROG2 se despiertan **
** PROG3 - termine de ejecutar **
PROG1 - me despierto y ejecuto 999999 ciclos
Chau soy PROG1
PROG2 - me despierto y ejecuto 5999999 ciclos
Chau soy PROG2
Chau soy PROG3

Cmd>
| ayuda | ps  |  init | mem | segs  | cls   | F2: Log | Mouse: è/%4C4C+)*4613
IPS: 14.001M            A:  NUM  CAPS  SCRL  UHCI  OHCI
```

[Fig. 4]

Conclusión

El algoritmo SJF es un algoritmo eficiente ya que reduce fuertemente el tiempo de espera, pero como contrapartida se tiene como desventaja el manejo de las estadísticas, ya que este requiere un alto costo para el guardado de los historiales de ejecución para cada proceso, y además el cálculo para estimar la duración de la próxima ráfaga de CPU.

Propuesta de mejoras

Actualmente todo el cálculo de las estadísticas se realiza sólo cuando el algoritmo SJF está activo podría evaluarse, si vale la pena, que estas se lleven a cabo en todo momento, sobre todo para mantener cierta coherencia cuando se hacen los cambios de algoritmo en tiempo de ejecución.

Un problema no menor que tuvimos se origina debido a que las estadísticas se asocian a los nombres de los binarios y cuando se crea un proceso nuevo el mismo se genera clonando SODSHELL y luego remplazando la imagen de este por el del proceso nuevo. El inconveniente es que cuando se ejecuta el planificador, previo al remplazo de la imagen del proceso, se selecciona al nuevo proceso pero en base a las estadísticas recopiladas para “SODSHELL”. Luego, al cambiar la imagen del proceso se actualizan las estadísticas del mismo pero este ya se encuentra en el procesador por lo que el planificador nunca llega a seleccionarlo porque sea el “proceso más corto”.

Lo mencionado anteriormente es una limitación cuando no existe ningún proceso activo en SODIUM fuera de IDLE, INIT y SODSHELL. Sin embargo, las pruebas que realizamos fueron exitosas ya que durante la ejecución el sistema operativo tiene varios procesos listos y en estas

condiciones el planificador sí dispone de las estadísticas correctas para seleccionar al proceso más corto.

Las posibles soluciones que consideramos para este inconveniente fueron:

- Lograr que la ejecución de fork y execve al ejecutar un proceso nuevo sea atómica. Esto, aun siendo una solución definitiva a este problema, consideramos que no es viable para implementar.

- Modificar la función fork y agregarle un parámetro no obligatorio que le indique el nombre del binario por el que se remplazará el nuevo proceso. La finalidad de esta modificación es que a la hora de cargar las estadísticas en el PCB del proceso se seleccionen las estadísticas del binario recibido por parámetro y no las del proceso que está siendo duplicado.

Finalmente, se debe tener en cuenta que dada la imposibilidad de registrar las estadísticas de los binarios en medios físicos, sino que estas solo se conservan en memoria principal, se recomienda realizar algunas ejecuciones de los procesos a probar lo que proveerá al sistema operativo de estadísticas más consistentes para mejorar el criterio de selección.

Estadísticas de los procesos y de la CPU

Introducción

Las estadísticas a analizar, pueden separase en dos grupos: procesos y CPU. Las primeras, hacen referencia a información relacionada a cada uno de los procesos que se pueden ejecutar en el sistema operativo, es decir, que cada programa tendrá sus propias estadísticas, basadas en su historial de ejecuciones. Por otro lado, el sistema operativo tendrá sus propias estadísticas, las cuales dependerán no solo de las características del sistema operativo en sí, sino de los procesos que se ejecuten en él y como logre este dar respuesta a los mismo, en base a los recursos de los que dispone

Resumen de modificaciones y agregados

Variables

Archivo: /include/kernel/pcb.h

Variable: Se agrega a la estructura *_stuPCB* la variable *sjfPrimeraRafaga*

Objetivo: El propósito es almacenar en el PCB del proceso información sobre tiempo de inicio del proceso:

```
typedef struct _stuPCB_ {
…
unsigned long sjfPrimeraRafaga;/*!<Instante en que comienza la primera
rafaga*/
…
```

Archivo: /include/kernel/gdt.h

Objetivo: Se agregan variables a la estructura de datos, que se utilizarán para almacenar las estadísticas.

```
typedef struct _pStat_
{
…
unsigned long turnaroundPromedio;
unsigned long turnaroundAcumulado;
unsigned long waitingPromedio;
unsigned long waitingAcumulado;
unsigned long responsePromedio;
unsigned long responseAcumulado;
…
} pStat;
```

Funciones

Archivo: usr/usuario.c
Función: main()
Objetivo: Se agrega la opción para mostrar las estadísticas.

```
...
 case ('S'):
    vFnImprimir("\nEstadisticas");
    showstat();
    break;                    {
...
```

Archivo: usr/lib/libsodium.c
Función: showstat()
Objetivo: Se agrega la llamada al syscall para acceder a las estadísticas.

```
long showstat()

{
  long liRetorno;
  SYS_CALL_0( liRetorno, errno, __NR_showstat );
  return liRetorno;
}
```

Archivo: kernel/syscall.c
Función: lFnSysShowStat()
Objetivo: Se recorre el vector de estadísticas y se imprimen las mismas.

```
long lFnSysShowStat()
{
  int i=0;
  for(i=0;i<CANTMAXPROCS;i++)
    if(pstuPCB[i].iEstado != -1)
    {
      vFnImprimir("...");
    }
  int uiIndiceVectorStat;
  for(uiIndiceVectorStat=0;
uiIndiceVectorStat<CANTMAXSTATPROCS;  uiIndiceVectorStat++
)
  {
    if(procStats[uiIndiceVectorStat].libre == 'N')
    {
      vFnImprimir("...");
    }
  }
  vFnImprimir ("\n Fin estadisticas \n");
  return 0;
}
```

Modo de ejecución

Desde la línea de comandos se debe ejecutar el comando “usuario S”. Esto traerá la lista de estadísticas correspondientes para cada proceso y la CPU.