



CENTRO DE ESTUDIOS FINANCIEROS

VIRIATO, 52	28010 MADRID	914 44 49 20
PONZANO, 15	28010 MADRID	914 44 49 20
G. DE GRÀCIA, 171	08012 BARCELONA	934 15 09 88
ALBORAYA, 23	46010 VALENCIA	963 61 41 99

www.cef.es

info@cef.es

Índice Tema 9

1. Control de ejecución de los jobs.
 - 1.1. Concepto de «job control».
 - 1.2. Acceso al terminal de control.
 - 1.3. Grupos de Proceso huérfanos.
 - 1.4. Desarrollo de una «shell» de Control de Job.
 - 1.5. Estructuras de Datos para la shell.
 - 1.6. Inicializar la shell.
 - 1.7. Lanzamiento de jobs.
 - 1.8. Jobs parados y terminados.
 - 1.9. Rearranque de jobs parados.
2. Evaluación del rendimiento. Planificación de la capacidad. Análisis de la carga. Herramientas y técnicas utilizables.





CENTRO DE ESTUDIOS FINANCIEROS

VIRIATO, 52	28010 MADRID	914 44 49 20
PONZANO, 15	28010 MADRID	914 44 49 20
G. DE GRÀCIA, 171	08012 BARCELONA	934 15 09 88
ALBORAYA, 23	46010 VALENCIA	963 61 41 99

www.cef.es

info@cef.es

TEMA 9

Control de ejecución de los trabajos. Evaluación del rendimiento. Planificación de la capacidad. Análisis de la carga. Herramientas y técnicas utilizables.

1. CONTROL DE EJECUCIÓN DE LOS JOBS.

El control de la ejecución de los jobs se realiza vía el «Job Control». Hemos visto en el tema anterior dicho «Job Control» referido a máquinas trabajando en entornos «mainframe», en este caso vamos a verlo desde el punto de vista de máquinas trabajando con UNIX o con LINUX.

1.1. CONCEPTOS DE «JOB CONTROL».

El objetivo fundamental de una shell interactiva es leer comandos desde el terminal del usuario y crear procesos para ejecutar los programas especificados por estos comandos. Para ello se utilizan los comandos «fork» (Para crear un proceso) y «exec» (Para ejecutar un fichero).

Un simple comando puede ejecutar un proceso y muy a menudo un comando utiliza varios procesos. Por ejemplo una simple compilación utiliza cuatro procesos.

Los procesos que pertenecen a un único comando se conocen como un grupo de procesos o un job. De esta manera se puede operar con todos estos procesos a la vez. Por ejemplo, tecleando C-c envía la señal SIGINT para terminar todos los procesos en el grupo de procesos en foreground. Una sesión es el grupo más grande de procesos. Normalmente todos los procesos que se generan en un solo login pertenecen a la misma sesión.

Cada proceso pertenece a un grupo de procesos. Cuando se crea un proceso, se convierte en un miembro del mismo grupo de procesos y sesión que el del proceso padre. Se puede cambiar de grupo de procesos empleando la función «setpgid» siempre que el grupo de procesos pertenezca a la misma sesión.

La única manera de colocar un proceso en una sesión diferente es convertir este proceso como proceso inicial de una nueva sesión, conocida como sesión líder, empleando la función «setsid».

Normalmente las nuevas sesiones son creadas por el programa de login y la sesión líder es el proceso ejecutando la shell de login. A continuación se ejecuta una shell que soporta el control de los jobs que utilizan los terminales en un momento dado.

La shell puede dar acceso ilimitado para controlar que un terminal acceda sólo a un grupo de procesos en un momento dado. Este tipo de job se conoce como un proceso «Foreground». Otros procesos que se ejecutan sin necesitar acceder a los terminales se conocen como «Background».

Es responsabilidad de la shell avisar al usuario cuando se para un job, así como prever mecanismos para que el usuario interactivamente pueda rearrancar jobs parados así como conmutar los jobs de foreground a background y viceversa.

No todos los sistemas operativos soportan «Job Control». El sistema GNU lo soporta pero si se utiliza la librería GNU de algún otro sistema puede que este otro sistema no permita la utilización de dicho «Job Control». Se puede utilizar la macro `_POSIX_JOB_CONTROL` para testear en tiempo de compilación si el sistema citado soporta «Job Control».

Uno de los atributos de un proceso es su terminal de control. Los procesos hijos creados con el comando `FORK` heredan el terminal de control de su proceso padre. De este modo, todos los procesos de una sesión heredan el terminal de control del líder de sesión.

1.2. ACCESO AL TERMINAL DE CONTROL.

Los procesos en un job de foreground de un terminal de control tienen el acceso sin restricción a dicho terminal, los procesos de background no lo hacen. Cuando un proceso foreground trata de leer de su terminal de control, al grupo de proceso por lo general se le envía una señal de `SIGTTIN`. Esto normalmente da lugar a que todos los procesos en ese grupo se paren (a no ser que ellos manejen la señal y no se paren). Asimismo cuando un proceso foreground trata de escribir a su terminal de control, el comportamiento por defecto es enviar una señal de `SIGTTOU` al grupo de proceso.

1.3. GRUPOS DE PROCESO HUÉRFANOS.

Cuando un proceso de control se termina, el terminal queda disponible y una nueva sesión puede ser establecida (otro usuario podría conectarse sobre el terminal). Esto podría causar un problema si cualquier proceso de la sesión anterior todavía trata de usar dicho terminal. Para prevenir estos problemas los grupos de procesos que siguen ejecutándose después de que la sesión haya terminado son marcados como grupos de proceso huérfanos. Cuando un grupo de proceso queda como huérfano, a sus procesos le son enviados una señal de `SIGHUP`. Generalmente, esto hace que los procesos se terminen. Sin embargo, si un programa no hace caso de esta señal o trata de manejarlo puede seguir ejecutándose como un proceso huérfano; pero no puede tener acceso nunca más al terminal.

1.4. DESARROLLO DE UNA «SHELL» DE CONTROL DE JOB.

1. Estructuras de Datos Introducción a la «shell» ejemplo.
2. Inicializar la «Shell»: cómo la «shell» efectúa el control de job.

3. Lanzamiento de Jobs: creación de jobs para ejecutar comandos.
4. Foreground y Background: puesta de un job en Foreground.
5. Jobs parados y jobs terminados: informe del estado de un job.
6. Cómo lanzar jobs parados.

1.5. ESTRUCTURAS DE DATOS PARA LA SHELL.

Todos los ejemplos incluidos en este tema son parte de un programa de shell. Se presentan estructuras de datos y las funciones de utilidad que se utilizan.

La shell del ejemplo muestra principalmente dos estructuras de datos. El tipo de job contiene la información sobre un job. El tipo de proceso mantiene la información sobre un solo subproceso. Aquí están las declaraciones de estructura de datos relevantes:

```
/* A process is a single process. */
typedef struct process
{
    struct process *next;           /* next process in pipeline */
    char **argv;                   /* for exec */
    pid_t pid;                     /* process ID */
    char completed;                /* true if process has completed */
    char stopped;                  /* true if process has stopped */
    int status;                    /* reported status value */
} process;
/* A job is a pipeline of processes. */
typedef struct job
{
    struct job *next;              /* next active job */
    char *command;                 /* command line, used for messages */
    process *first_process;        /* list of processes in this job */
    pid_t pgid;                    /* process group ID */
    char notified;                 /* true if user told about stopped job */
    struct termios tmodes;         /* saved terminal modes */
    int stdin, stdout, stderr;     /* standard i/o channels */
} job;
/* The active jobs are linked into a list. This is its head. */
job *first_job = NULL;
```

• Utilidades para manejar el objeto JOB.

```
/* Find the active job with the indicated pgid. */
job *
find_job (pid_t pgid)
{
    job *j;

    for (j = first_job; j; j = j->next)
```

```

        if (j->pgid == pgid)
            return j;
    return NULL;
}

/* Return true if all processes in the job have stopped or completed. */
int
job_is_stopped (job *j)
{
    process *p;

    for (p = j->first_process; p; p = p->next)
        if (!p->completed && !p->stopped)
            return 0;
    return 1;
}

/* Return true if all processes in the job have completed. */
int
job_is_completed (job *j)
{
    process *p;

    for (p = j->first_process; p; p = p->next)
        if (!p->completed)
            return 0;
    return 1;
}

```

1.6. INICIALIZAR LA SHELL.

Una subshell que se ejecuta interactivamente tiene que asegurar que ha sido colocada en el foreground por su shell padre antes de permitir el control de job en sí mismo. Esto se hace consiguiendo su grupo de proceso inicial ID con la función de `getpgrp`, y comparándolo al grupo de proceso ID del job de foreground en curso asociado con su terminal de control (que puede ser recuperado usando la función de `tgetpgrp`). Si la subshell no se ejecuta como un job de foreground debe pararse enviando una señal de `SIGTTIN` a su propio grupo de proceso. Esto arbitrariamente no puede ponerse en foreground, debe esperar al usuario para decir a la shell padre que lo haga.

Una vez que la subshell ha sido colocada en el foreground por su shell padre, entonces puede permitir su propio control de job, se hace llamando a `setpgid` para ponerse en su propio grupo de proceso y luego llamando a `tcsetpgrp` para colocar este grupo de proceso en el foreground. Cuando una shell permite el control de job debe tener en cuenta todos los stops de control de job de modo que no se pare por casualidad.

Una subshell que se no ejecuta interactivamente no puede efectuar el control de jobs, debe dejar todos los procesos que crea en el mismo grupo de proceso de la shell, esto permite que la shell no interactiva y sus procesos hijos sean tratadas como un job sólo por la shell padre.

Aquí está el código de inicialización para la shell del ejemplo.

```

/* Keep track of attributes of the shell. */

#include <sys/types.h>
#include <termios.h>
#include <unistd.h>

pid_t shell_pgid;
struct termios shell_tmodes;
int shell_terminal;
int shell_is_interactive;

/* Make sure the shell is running interactively as the foreground job
before proceeding. */

void
init_shell ()
{
    /* See if we are running interactively. */
    shell_terminal = STDIN_FILENO;
    shell_is_interactive = isatty (shell_terminal);

    if (shell_is_interactive)
    {
        /* Loop until we are in the foreground. */
        while (tcgetpgrp (shell_terminal) != (shell_pgid = getpgrp ()))
            kill (- shell_pgid, SIGTTIN);

        /* Ignore interactive and job-control signals. */
        signal (SIGINT, SIG_IGN);
        signal (SIGQUIT, SIG_IGN);
        signal (SIGTSTP, SIG_IGN);
        signal (SIGTTIN, SIG_IGN);
        signal (SIGTTOU, SIG_IGN);
        signal (SIGCHLD, SIG_IGN);

        /* Put ourselves in our own process group. */
        shell_pgid = getpid ();
        if (setpgid (shell_pgid, shell_pgid) < 0)
        {
            perror ("Couldn't put the shell in its own process group");
            exit (1);
        }

        /* Grab control of the terminal. */
        tcsetpgrp (shell_terminal, shell_pgid);

        /* Save default terminal attributes for shell. */
        tcgetattr (shell_terminal, &shell_tmodes);
    }
}

```

1.7. LANZAMIENTO DE JOBS.

Una vez que la shell ha tomado la responsabilidad de realizar el control de job sobre su terminal de control, puede lanzar jobs en respuesta a comandos escritos por el usuario. Para crear los procesos

en un grupo de proceso se usa el mismo comando FORK y funciones de EXEC. Como hay múltiples procesos hijos implicados las cosas son un poco más complicadas y se debe procurar hacer las cosas en el orden correcto.

Hay dos opciones para estructurar el árbol de relaciones padre hijos entre los procesos. Se puede hacer que todos los procesos en el grupo de proceso sean los hijos del proceso de la shell o que un proceso en el grupo sea el antepasado de los otros procesos en ese grupo. El programa de shell del ejemplo usa el primer acercamiento porque hace la contabilidad más simple. Con cada proceso bifurcado (forked) debería ponerse en el nuevo grupo de proceso llamando a setpgid. El primer proceso en el nuevo grupo se convierte en su líder de grupo de proceso y su proceso ID se convierte en el grupo de proceso ID para el grupo.

La shell también debe llamar a setpgid para poner a cada uno de sus procesos hijos en el nuevo grupo de proceso ya que hay un problema de cronometraje potencial, cada proceso hijo debe ser puesto en el grupo de proceso antes de que comience a ejecutar un nuevo programa y la shell debe tener todos sus procesos hijos en el grupo antes de que siga ejecutando. Si tanto los procesos hijos como la shell llaman a setpgid, esto asegura que las cosas suceden como debe ser. Si el job está siendo lanzado como un job de foreground el nuevo grupo de proceso también tiene que ser puesto en el foreground sobre el terminal de control que usa tcsetpgrp. Otra vez, esto debe ser hecho por la shell así como cada uno de sus procesos hijos. Lo siguiente que cada proceso hijo debe hacer es recomponer sus acciones de señal.

Durante la inicialización el proceso de shell ignora las señales de control de job, por consiguiente cualquier proceso hijo que crea tampoco hace caso de estas señales. Como esto es indeseable entonces cada proceso hijo debe colocar las señales atrás después de que es bifurcado (forked).

Ya que las shells siguen esta convención las aplicaciones heredan el manejo correcto de estas señales del proceso padre. Pero cada aplicación tiene una responsabilidad de no estropear el manejo de stops. Las aplicaciones que incapacitan la interpretación normal del carácter SUSP debe proporcionar otro mecanismo para parar el job. Cuando el usuario invoca este mecanismo, el programa debería enviar una señal de SIGTSTP al grupo de proceso del proceso no solamente al proceso en sí mismo.

Finalmente cada proceso hijo debe llamar al comando exec. La función del programa de shell del ejemplo es responsable de lanzar un programa. La función es ejecutada por cada proceso hijo inmediatamente después de que ha sido bifurcado por la shell y nunca vuelve.

```
void
launch_process (process *p, pid_t pgid,
                int infile, int outfile, int errfile,
                int foreground)
{
    pid_t pid;
    if (shell_is_interactive)
    {
        /* Put the process into the process group and give the process group
           the terminal, if appropriate.
           This has to be done both by the shell and in the individual
           child processes because of potential race conditions. */
        pid = getpid ();
```



```

        if (pgid == 0) pgid = pid;
        setpgid (pid, pgid);
        if (foreground)
            tcsetpgrp (shell_terminal, pgid);

        /* Set the handling for job control signals back to the default.
        */
        signal (SIGINT, SIG_DFL);
        signal (SIGQUIT, SIG_DFL);
        signal (SIGTSTP, SIG_DFL);
        signal (SIGTTIN, SIG_DFL);
        signal (SIGTTOU, SIG_DFL);
        signal (SIGCHLD, SIG_DFL);
    }

    /* Set the standard input/output channels of the new process. */
    if (infile != STDIN_FILENO)
    {
        dup2 (infile, STDIN_FILENO);
        close (infile);
    }
    if (outfile != STDOUT_FILENO)
    {
        dup2 (outfile, STDOUT_FILENO);
        close (outfile);
    }
    if (errfile != STDERR_FILENO)
    {
        dup2 (errfile, STDERR_FILENO);
        close (errfile);
    }

    /* Exec the new process. Make sure we exit. */
    execvp (p->argv[0], p->argv);
    perror ("execvp");
    exit (1);
}

```

Si la shell no se ejecuta interactivamente, esta función no hace nada con grupos de proceso o señales. Recuerde que una shell que no realiza el control de job debe mantener todos sus subprocesos en el mismo grupo de proceso que la shell misma.

Después está la función que en realidad lanza un job completo. Después de la creación de los procesos hijos, esta función llama algunas otras funciones para poner el job recién creado en el foreground o el background.

```

void
launch_job (job *j, int foreground)
{
    process *p;
    pid_t pid;
    int mypipe[2], infile, outfile;

```

```

infile = j->stdin;
for (p = j->first_process; p; p = p->next)
{
    /* Set up pipes, if necessary. */
    if (p->next)
    {
        if (pipe (mypipe) < 0)
        {
            perror ("pipe");
            exit (1);
        }
        outfile = mypipe[1];
    }
    else
        outfile = j->stdout;

    /* Fork the child processes. */
    pid = fork ();
    if (pid == 0)
        /* This is the child process. */
        launch_process (p, j->pgid, infile,
                        outfile, j->stderr, foreground);
    else if (pid < 0)
    {
        /* The fork failed. */
        perror ("fork");
        exit (1);
    }
    else
    {
        /* This is the parent process. */
        p->pid = pid;
        if (shell_is_interactive)
        {
            if (!j->pgid)
                j->pgid = pid;
            setpgid (pid, j->pgid);
        }
    }

    /* Clean up after pipes. */
    if (infile != j->stdin)
        close (infile);
    if (outfile != j->stdout)
        close (outfile);
    infile = mypipe[0];
}

format_job_info (j, "launched");
if (!shell_is_interactive)
    wait_for_job (j);
else if (foreground)
    put_job_in_foreground (j, 0);

```

```

else
    put_job_in_background (j, 0);
}

```

Ahora vamos a considerar qué acciones deben ser tomadas por la shell cuando lanza un job al foreground y cómo se diferencia cuando se lanza un job de background.

Cuando se lanza un job de foreground la shell primero debe dar acceso al terminal de control llamando a `tcsetpgrp`. Entonces la shell debe esperar a que los procesos en aquel grupo de proceso se terminen o paren.

Cuando todos los procesos en el grupo han terminado o han parado, la shell debe recuperar el control del terminal para su propio grupo de proceso llamando a `tcsetpgrp` otra vez. El job de foreground puede haber dejado al terminal en un estado extraño entonces la shell debe restaurar la situación anterior. En caso de job simplemente parado la shell primero debería salvar el estado del terminal de modo que pueda restaurarlo más tarde si reanuda el job. Las funciones para tratar con estas situaciones de terminales son `tcgetattr` y `tcsetattr`.

La función de la shell del ejemplo para hacer esto es:

```

/* Put job j in the foreground. If cont is nonzero,
   restore the saved terminal modes and send the process group a
   SIGCONT signal to wake it up before we block. */

void
put_job_in_foreground (job *j, int cont)
{
    /* Put the job into the foreground. */
    tcsetpgrp (shell_terminal, j->pgid);

    /* Send the job a continue signal, if necessary. */
    if (cont)
    {
        tcsetattr (shell_terminal, TCSADRAIN, &j->tmodes);
        if (kill (-j->pgid, SIGCONT) < 0)
            perror ("kill (SIGCONT)");
    }

    /* Wait for it to report. */
    wait_for_job (j);

    /* Put the shell back in the foreground. */
    tcsetpgrp (shell_terminal, shell_pgid);

    /* Restore the shell's terminal modes. */
    tcgetattr (shell_terminal, &j->tmodes);
    tcsetattr (shell_terminal, TCSADRAIN, &shell_tmodes);
}

```

Si el grupo de proceso se lanza como un job de background la shell debe permanecer en el foreground y seguir leyendo comandos del terminal.

La shell del ejemplo coloca un job en el background, la función que usa es:

```
/* Put a job in the background. If the cont argument is true, send
   the process group a SIGCONT signal to wake it up. */

void
put_job_in_background (job *j, int cont)
{
    /* Send the job a continue signal, if necessary. */
    if (cont)
        if (kill (-j->pgid, SIGCONT) < 0)
            perror ("kill (SIGCONT)");
}
```

1.8. JOBS PARADOS Y TERMINADOS.

Cuando se lanza un proceso de foreground la shell debe bloquearse hasta que todos los procesos en aquel job estén o terminados o parados, y puede hacerse llamando a la función waitpid.

La shell también debe comprobar el estado de jobs de background de modo que pueda informar de los jobs terminados y parados al usuario, se puede hacer llamando al comando waitpid con la opción WNOHANG.

La shell también puede recibir la notificación asincrónica de que hay información de estado disponible para un proceso hijo estableciendo un manejador para señales de SIGCHLD.

En el programa de shell del ejemplo, a la señal de SIGCHLD normalmente no se le hace caso. Esto es para evitar problemas de reentrada relacionados con los datos globales que la estructura de la shell maneja.

La shell del ejemplo comprueba el estado de jobs y produce un informe al usuario.

```
/* Store the status of the process pid that was returned by waitpid.
   Return 0 if all went well, nonzero otherwise. */

int
mark_process_status (pid_t pid, int status)
{
    job *j;
    process *p;

    if (pid > 0)
    {
        /* Update the record for the process. */
        for (j = first_job; j; j = j->next)
            for (p = j->first_process; p; p = p->next)
                if (p->pid == pid)
                {
                    p->status = status;
                }
    }
}
```

```

        if (WIFSTOPPED (status))
            p->stopped = 1;
        else
        {
            p->completed = 1;
            if (WIFSIGNALED (status))
                fprintf (stderr, "%d: Terminated by signal %d.\n",
                        (int) pid, WTERMSIG (p->status));
        }
        return 0;
    }
    fprintf (stderr, "No child process %d.\n", pid);
    return -1;
}
else if (pid == 0 || errno == ECHILD)
    /* No processes ready to report. */
    return -1;
else {
    /* Other weird errors. */
    perror ("waitpid");
    return -1;
}
}

/* Check for processes that have status information available,
without blocking. */

void
update_status (void)
{
    int status;
    pid_t pid;

do
    pid = waitpid (WAIT_ANY, &status, WUNTRACED|WNOHANG);
while (!mark_process_status (pid, status));
}

/* Check for processes that have status information available,
blocking until all processes in the given job have reported. */

void
wait_for_job (job *j)
{
    int status;
    pid_t pid;

do
    pid = waitpid (WAIT_ANY, &status, WUNTRACED);
while (!mark_process_status (pid, status)
        && !job_is_stopped (j)
        && !job_is_completed (j));
}

```

```

/* Format information about job status for the user to look at. */

void
format_job_info (job *j, const char *status)
{
    fprintf (stderr, "%ld (%s): %s\n", (long)j->pgid, status, j->command);
}

/* Notify the user about stopped or terminated jobs.
   Delete terminated jobs from the active job list. */

void
do_job_notification (void)
{
    job *j, *jlast, *jnext;
    process *p;

/* Update status information for child processes. */
update_status ();

jlast = NULL;
for (j = first_job; j; j = jnext)
    {
        jnext = j->next;

        /* If all processes have completed, tell the user the job has
           completed and delete it from the list of active jobs. */
        if (job_is_completed (j)) {
            format_job_info (j, "completed");
            if (jlast)
                jlast->next = jnext;
            else
                first_job = jnext;
            free_job (j);
        }

        /* Notify the user about stopped jobs,
           marking them so that we won't do this more than once. */
        else if (job_is_stopped (j) && !j->notified) {
            format_job_info (j, "stopped");
            j->notified = 1;
            jlast = j;
        }

        /* Don't say anything about jobs that are still running. */
        else
            jlast = j;
    }
}

```

1.9. REARRANQUE DE JOBS PARADOS.

La shell puede controlar un job parado enviando una señal de SIGCONT a su grupo de procesos. Si el job está siendo controlado en el foreground la shell debe invocar a tcsetpgrp para dar acceso de

job al terminal y restaurar los ajustes salvados de los terminales. Después del re arranque de un job en el foreground la shell debe esperar a que el job se pare o se complete como si el job hubiese sido lanzado en el foreground.

El programa de shell del ejemplo maneja tantos jobs recién creados como re arrancados con el mismo par de funciones, `put_job_in_foreground` y `put_job_in_background`.

```
/* Mark a stopped job J as being running again. */
```

```
void
mark_job_as_running (job *j)
{
    Process *p;

    for (p = j->first_process; p; p = p->next)
        p->stopped = 0;
    j->notified = 0;
}
```

```
/* Continue the job J. */
```

```
void
continue_job (job *j, int foreground)
{
    mark_job_as_running (j);
    if (foreground)
        put_job_in_foreground (j, 1);
    else
```

2. EVALUACIÓN DEL RENDIMIENTO. PLANIFICACIÓN DE LA CAPACIDAD. ANÁLISIS DE LA CARGA. HERRAMIENTAS Y TÉCNICAS UTILIZABLES.

La SPEC (Standard Performance Evaluation Corporation) es una corporación no lucrativa creada para establecer, mantener y aprobar un juego estandarizado de «benchmark» (pruebas patrón) que pueden ser aplicados a la última generación de ordenadores de alto rendimiento. La SPEC también publica los resultados de dichas pruebas.

La relación de pruebas realizadas por SPEC vienen indicadas en:

webmaster@spec.org

Last updated: Fri Jan 23 15:41:28 EST 2004

Copyright © 1995 - 2004 Standard Performance Evaluation Corporation

URL: <http://www.spec.org/benchmarks.html>

Vamos a presentar un resumen por áreas informáticas de las pruebas realizadas:

1. CPU.

CPU2000 Diseñado para proporcionar medidas de resultados para poder efectuar comparaciones. Calculan cargas de job intensivas sobre ordenadores diferentes, SPEC CPU2000 contiene dos suites de pruebas: CINT2000 para medir y comparar rendimientos para tratamientos intensivos de número entero, y CFP2000 para medir y comparar rendimientos para tratamientos intensivos de coma flotante.

2. Servicios para la empresa.

appPlatform Actualmente en desarrollo, SPECappPlatform se ha diseñado para medir la escalabilidad y el funcionamiento de las plataformas de empresa (como J2EE y .NET) durante la ejecución de servicios web. La prueba (benchmark) incluirá las características que son típicas en las aplicaciones de empresa, como transacciones, servicios de persistencia, servicios de web, y de mensajería.

3. Aplicaciones Gráficas.

La relación de benchmark existentes las relacionamos a continuación:

- SPECviewperf® 7.1.1
- SPECviewperf® 7.1
- SPECapcSM for 3ds maxTM 4.2
- SPECapcSM for Maya 5
- SPECapcSM for Pro/ENGINEERTM 2001
- SPECapcSM for Solid Edge V12TM
- SPECapcSM for SolidWorks 2003
- SPECapcSM for Unigraphics V17

4. Informática de Alto rendimiento, OpenMP, MPI.

HPC2002 Mide el funcionamiento de los sistemas de alto rendimiento que controlan aplicaciones industriales y sobre todo es útil para evaluar el funcionamiento de arquitecturas de ordenador paralelas y distribuidas.

OMP2001 Mide el funcionamiento de los sistemas de alto rendimiento que controlan aplicaciones basadas en el estándar OpenMP para el proceso paralelo en memoria compartida. Para ello hay dos herramientas OMPM2001 y OMPL2001.

5. Servidores/Clientes Java.

jAppServer2002 SPECjAppServer2002 es similar al benchmark SPECjAppServer2001, pero el código de servidor de aplicación ha sido migrado para utilizar el estándar J2EE 1.3.

jAppServer2001 Es un benchmark de cliente/servidor para medir el funcionamiento de Servidores Java de aplicaciones empresariales que utilizan un subconjunto de la API J2EE.

JBB2000 Es un benchmark para evaluar el funcionamiento de servidores que controlan aplicaciones de gestión en Java. El benchmark puede emplearse para evaluar el funcionamiento de hardware y el software de la Máquina Java Virtual (JVM).

JVM98 SPECjvm98 permite a los usuarios evaluar el funcionamiento del hardware y del software de la plataforma cliente JVM. En el software, mide la eficacia de JVM, el compilador justo a tiempo (JIT), y la puesta en práctica del sistema operativo. En el hardware, la CPU (el número entero y el punto flotante), la cache, la memoria, y otros aspectos del funcionamiento específico de la plataforma.

jAppServer2003 Actualmente en desarrollo, jAppServer2003 se ha diseñado para medir el funcionamiento de los servidores de aplicación J2EE 1.3.

6. Servidores de Correo.

MAIL2001 Es un benchmark de servidor de correo estandarizado para medir la capacidad de un sistema de actuar como servidor de correo que las peticiones de correo electrónico basadas en los protocolos de estándar de Internet SMTP y POP3. El benchmark mide el rendimiento y tiempo de respuesta de un sistema mailserver con conexiones de red realistas, almacenamiento en disco y cargas de job de cliente.

IMAP2003 Actualmente en desarrollo por la SPEC es un benchmark estándar de la industria diseñada para medir el funcionamiento de los servidores del correo electrónico corporativo. Este benchmark probará la capacidad de un servidor de tratar las peticiones del correo electrónico, basadas en los protocolos de estándar de Internet SMTP y IMAP4.

7. Network File System.

SFS97_R1 (3.0) El benchmark de la SPEC está diseñado para evaluar la velocidad y capacidad de tratamiento de las peticiones NFS.

8. Servidores Web.

WEB99 Es un benchmark cliente/servidor para medir el número máximo de las conexiones simultáneas que un servidor de web es capaz de soportar.

WEB99_SSL Estos benchmark aseguran el funcionamiento de servidor de web que usa HTTP 1.0/1.1 sobre el protocolo SSL.

WEB2004 Actualmente en el desarrollo, el benchmark consiste en probar con cargas de job diferentes (tanto SSL como non-SSL) con la banca y el comercio electrónico.

Los benchmark para HPC2002 son:

- La SPEC CHEM2002, que está basado en una aplicación de química cuántica llamada GAMESS y tiene la métrica de funcionamiento SPECchemM2002 SPECchemS2002.

- La SPEC ENV2002, que está basado en un modelo de investigación meteorológica y pronóstico llamado WRF. Tiene dos métricas de funcionamiento, SPECenvM2002 SPECenvS2002.
- La SPEC SEIS2002, que representa una aplicación industrial que se utiliza para localizar el gas y los yacimientos petrolíferos y tiene la métrica de funcionamiento SPECseisM2002 SPECseisS2002.

Documentación en:

SPEC HPC2002 Benchmark Documentation

- SPEC HPC2002 Run and Reporting Rules.
- SPEC CHEM2002.
- SPEC SEIS2002.
- SPEC ENV2002.

SPECjAppServer2002 (Servidor Java de aplicaciones) es un benchmark de cliente/servidor para medir el funcionamiento de los Servidores Java de aplicaciones empresariales que utilizan un subconjunto de la API J2EE en una aplicación web.

La SPEC ha diseñado el benchmark SPECjAppServer2002 para probar al Servidor Java de aplicaciones empresariales, así como la Máquina Java Virtual (JVM) y los Sistemas de servidor en la prueba (SUT).

SPECjAppServer2002 Benchmark Documentation

- SPECjAppServer2002 FAQ.
- SPECjAppServer2002 Support FAQ.
- SPECjAppServer2002 Run and Reporting Rules.
- SPECjAppServer2002 User's Guide.
- SPECjAppServer2002 Design Document.

SPECmail2001 es un benchmark de servidor de correo estandarizado para medir la capacidad de un sistema para actuar como servidor de correo que recibe peticiones de correo electrónico basadas en los protocolos de estándar de Internet SMTP y POP3. El benchmark estudia el rendimiento y el tiempo de respuesta de un sistema mailserver probando conexiones de red realistas, el almacenamiento en disco y cargas de job de cliente para una carga de aproximadamente 10.000 a 1.000.000 de usuarios. El objetivo es de permitir las comparaciones objetivas de productos de servidor de correo.

SPEC MAIL2001 Benchmark Documentation:

- SPC MAIL2001 FAQ.
- SPEC MAIL2001 User's Guide.
- SPEC MAIL2001 Support FAQ.
- SPEC MAIL2001 Run and Reporting Rules.
- SPEC MAIL2001 Result Submission Guidelines.
- SPEC MAIL2001 Design Document.

SFS97 R1 V3.0

SPEC introdujo en 1997 su nuevo SFS (el Servidor de ficheros de Sistema) el benchmark: SFS97, el reemplazo del original benchmark SFS conocido como LADDIS o SFS93 e incluye los siguientes rasgos principales:

- Corrige defectos detectados en SFS V2.0.
- Mide resultados tanto para la versión 3 de protocolo NFS como para la versión 2.
- Apoyo a TCP o UDP como transporte de red.
- El CD de distribución del benchmark incluye binarios precompilados y probados.
- Una interfaz acomodada tanto a usuarios principiantes como ya entrenados.
- Un instrumento de generación de informes.

SPECweb99 benchmark

SPECweb99 es el benchmark para evaluar el funcionamiento de Servidores de la World Wide Web. Como sucesor de SPECweb96, SPECweb99 sigue la tradición de SPEC de dar el benchmark más objetivo y representativo para los usuarios de Web para medir la capacidad de un sistema de actuar como servidor de web. En respuesta al avance fulminante de la tecnología de Web, el benchmark SPECweb99 incluye muchos detalles del «el estado del arte» para satisfacer las demandas de los usuarios de Web de hoy y mañana:

- Carga de job estandarizada, reconocida por las principales empresas del mercado de WWW.
- Medida de las conexiones simultáneas más bien que las operaciones HTTP.
- Simulación de conexiones con una velocidad de línea limitada.
- Keepalives (HTTP 1.0) y conexiones persistentes (HTTP 1.1).
- Rotación de anuncios dinámica que usa cookies y consultas.
- Una instalación automatizado para Windows NT de Microsoft así como para Unix.

SPECweb99 no soporta el uso de transacciones que utilizan el cifrado SSL. Para trabajar con SSL existe el benchmark SPECweb99_SSL.

SPECweb99 Documentation:

SPECweb99 FAQ.

SPECweb99 Support FAQ.

SPECweb99 Design Document.

SPECweb99 Benchmark Run and Reporting Rules.

SPECweb99 User's Guide.

• Benchmark existentes para pruebas de rendimiento.

TPC-C: La prueba de rendimiento TPC-C ha sido diseñada para medir la capacidad que tiene un servidor de funcionar como servidor de base de datos de procesamiento de transacciones en línea (OLTP). Esta prueba de rendimiento simula un ambiente de cómputo completo, en el que un grupo de usuarios ejecuta transacciones de pedidos en una base de datos. Estas transacciones incluyen colocar un nuevo pedido, efectuar o recibir un pago, revisar el estado del pedido, monitorear la entrega y verificar los niveles de inventario. El resultado TPC-C es una medición del número de nuevas transacciones de pedidos por minuto, a la vez que el sistema ejecuta los otros cuatro tipos de transacciones.

MMB2: La prueba de rendimiento de mensajes MAPI (MMB2) fue desarrollada por Microsoft para medir la capacidad que tiene un servidor de funcionar como una plataforma de servidor de Microsoft Exchange. MMB2 mide el rendimiento de un «usuario promedio» que ejecuta tareas comunes, tales como envío, vista, lectura y redireccionamiento de mensajes de e-mail, así como tareas de calendario y uso de listas de distribución durante un día laboral de 8 horas.

SPECweb@99: SPECweb99 ha sido diseñada para medir la capacidad que tiene un sistema de funcionar como un servidor Web de páginas Web dinámicas y estáticas. En la configuración de la prueba, un número de sistemas cliente genera solicitudes de páginas estáticas y dinámicas que se encuentran en un servidor, simulando una carga de job de servidor Web real. El resultado de la prueba SPECweb99 es el número máximo de conexiones simultáneas concordantes que un servidor Web es capaz de soportar, a la vez que cumple con requerimientos específicos de productividad e índices de error. Las conexiones concordantes deben mantenerse a un índice de bits máximo especificado con un tamaño de segmento máximo. Esto tiene como objetivo modelar condiciones reales que se experimentarán en Internet durante la duración de esta prueba.

SPECweb99_SSL: Al igual que SPECweb99, SPECweb99_SSL está diseñada para medir la capacidad que tiene un sistema de funcionar como un servidor Web para páginas estáticas y dinámicas. No obstante, SPECweb99_SSL difiere en que las páginas son solicitadas utilizando HTTP a través del protocolo seguro HTTPS. El resultado de la prueba tiene como objetivo medir la capacidad que tiene el sistema de funcionar como un servidor Web seguro.

SPECjbb™2000: SPECjbb2000 ha sido diseñada para medir la capacidad que tiene un sistema de funcionar como un servidor de aplicaciones Java de nivel intermedio en un sistema de 3

niveles. Este último consiste en un cliente (nivel 1), un servidor de aplicaciones Java (nivel 2) y una base de datos (nivel 3). El servidor de aplicaciones Java contiene la lógica de negocios y caches de objeto que manejan las solicitudes entrantes del cliente, recuperan la información apropiada de la base de datos y retornan esa información al cliente. Los servidores de aplicaciones Java se utilizan comúnmente en ERP, CRM, e-Business y otras aplicaciones de niveles, donde el navegador Web se utiliza para acceder información contenida en una base de datos. La prueba de rendimiento SPECjbb2000 está modelada con base en un mayorista con almacenes que prestan servicio a una cantidad de territorios. El resultado SPECjbb2000 mide la productividad de la plataforma Java, lo cual representa la velocidad en la que ciertas operaciones de negocios se ejecutan por segundo.

SPECint@2000: SPECint2000 fue diseñada para medir y comparar el rendimiento de cómputo de enteros entre sistemas. Esta prueba se enfoca en el rendimiento de:

- El procesador
- La arquitectura de la memoria
- Los compiladores

SPECint2000 consiste en 12 pruebas de rendimiento de enteros, las cuales fueron desarrolladas a partir de aplicaciones reales que utilizan los usuarios. Las aplicaciones con grandes volúmenes de enteros son las más comunes en los departamentos de informática e instalaciones de servidores empresariales. Las aplicaciones de bases de datos, servidores de e-mail, servidores de aplicaciones Java y servidores Web, por lo general, tienen mejor rendimiento cuando se ejecutan bajo un procesador con excelente ejecución de enteros.

SPECint_rate™2000: Esta prueba de rendimiento ejecuta el mismo algoritmo utilizado en SPECint2000, pero ejecuta múltiples instancias de la prueba en forma simultánea (por lo general, se ejecuta una instancia por procesador en el sistema). SPECint_rate2000 mide la capacidad que tiene un sistema de realizar múltiples operaciones de cómputo con grandes volúmenes de enteros a la vez. Esta prueba ha sido diseñada para medir la capacidad que tiene un sistema de multiprocesadores de escalar mientras ejecuta aplicaciones basadas en enteros, tales como los servidores de base de datos, servidores de e-mail o servidores Web.

SPECfp@2000: SPECfp2000 fue diseñada para medir y comparar el rendimiento de cómputo de punto flotante entre sistemas. Esta prueba se enfoca en el rendimiento de:

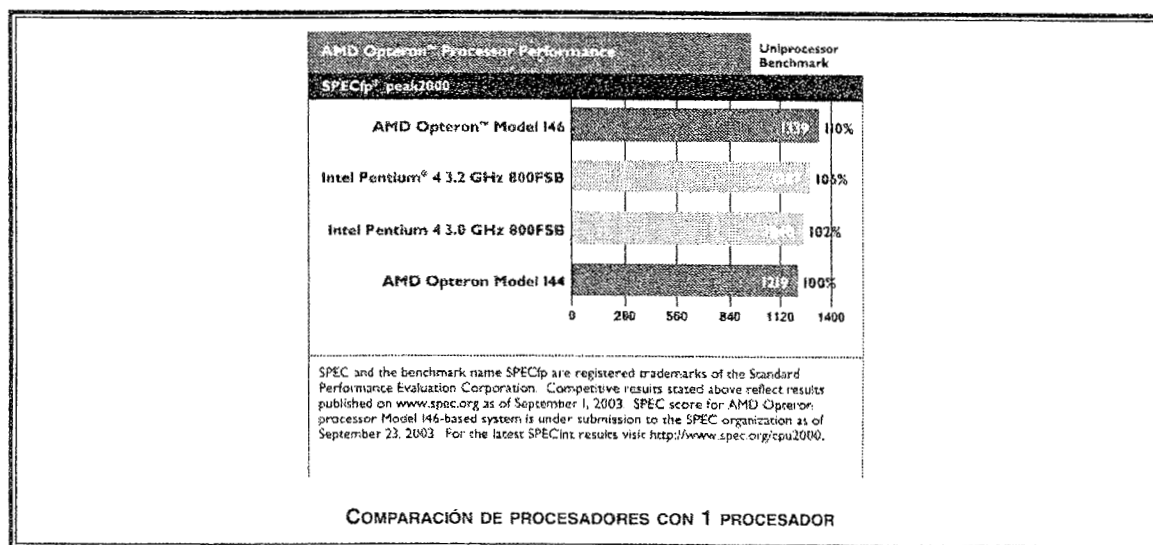
- El procesador
- La arquitectura de la memoria
- Los compiladores

SPECfp2000 consiste en 14 pruebas de punto flotante, las cuales fueron desarrolladas a partir de las aplicaciones que utilizan los usuarios en el ambiente de negocios real. Las aplicaciones con grandes requerimientos de punto flotante son las más comunes en los ambientes de ingeniería e investigación. Las aplicaciones, como las herramientas de dinámica de fluidos de cómputo, CAD/CAM, creación de contenido digital (DCC), reproducción y creación de modelos financieros, por lo general, tienen mejor rendimiento cuando se ejecutan en un procesador con excelente rendimiento de punto flotante.

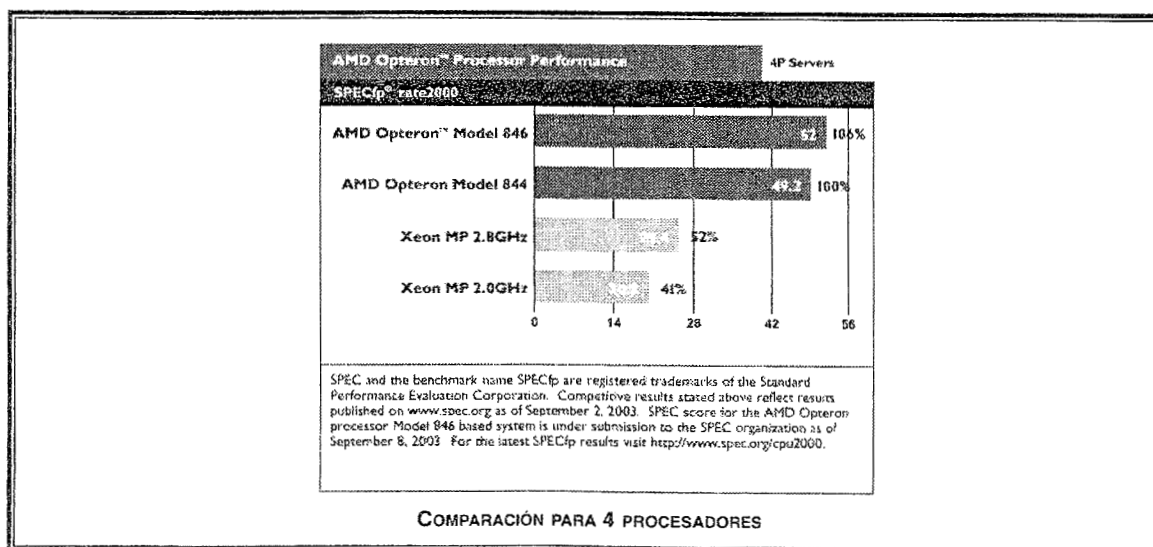
SPECfp_rate2000: Esta prueba de rendimiento ejecuta exactamente los mismos algoritmos utilizados en SPECfp2000, pero con la diferencia de que ejecuta múltiples instancias de la prueba a la vez (por lo general, se ejecuta una instancia por procesador en el sistema). SPECfp_rate2000 mide la capacidad que tiene un sistema de ejecutar múltiples operaciones de cómputo de punto flotante a la vez. Esta prueba ha sido diseñada para medir la capacidad que tienen un sistema de multiprocesadores de escalar mientras ejecuta aplicaciones basadas en punto flotante, como las aplicaciones CAD/CAM, DCC y otras de origen científico.

- Ejemplos donde se aplican los benchmark citados:

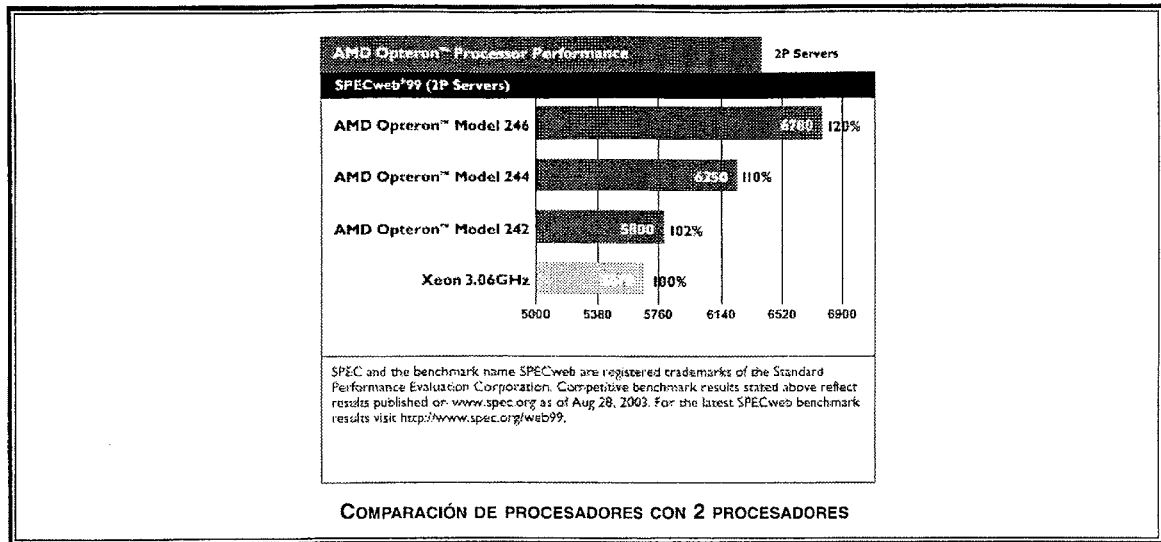
SPECfp®_peak2000 1P Servers



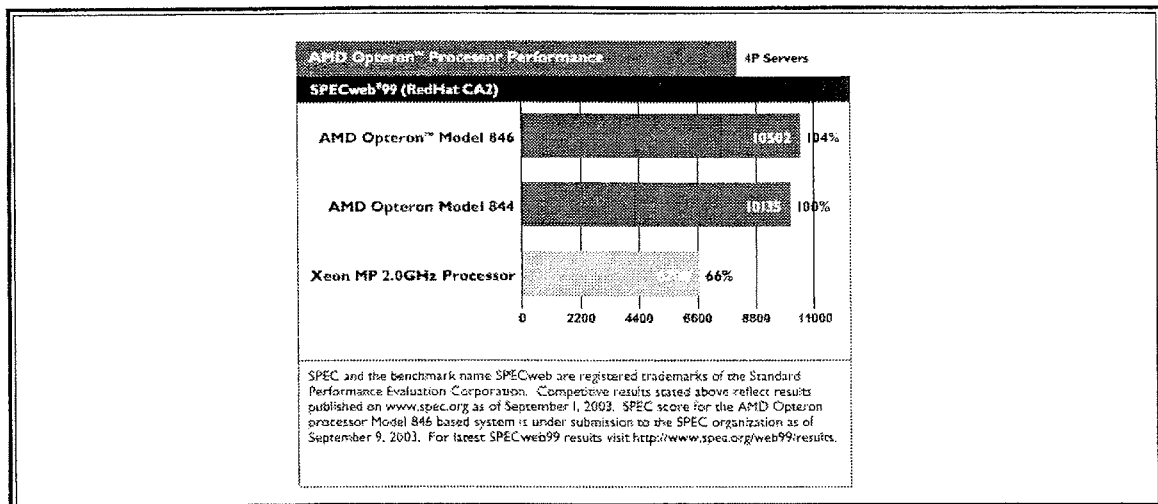
SPECfp®_rate2000 4P Servers



SPECweb®99 (Red Hat CA2) 2P Servers



SPECweb®99 (Red Hat CA2) 4P Servers



...

