Laboratorio 3: Planificador de procesos

Sistemas Operativos - FaMAF - UNC - 2024

Versión 2024: Milagro Teruel

Versión 2021-2023: Ignacio Moretti

Versiones 2014, 2016-2020: Carlos Bederián

Versión 2015: Facundo Ramallo, Pablo Ventura

Instalación

Deben seguir las mismas instrucciones que en el lab 2, pero sin aplicar los archivos patch.

1. **Clonar el repositorio original de xv6-riscv:**

```bash
$ git clone https://github.com/mit-pdos/xv6-riscv.git
$ cd xv6-riscv; git reset --hard f5b93ef
```

2. **Copiar los archivos clonados al repo de Bitbucket asignado a su grupo.**

```bash
git clone https://user@bitbucket.org/sistop-famaf/so24lab3gXX.git
cp -a xv6-riscv/* so24lab3gXX
rm -rf xv6-riscv
```

3. **Hacer el primer commit+push del proyecto**

```bash
cd so24lab3gXX
git tag -a original_code -m "Original xv6 code"
git add *  // Fijense que no hayan compilado código fuente. Hagan make clean
```

git commit -m 'Original xv6 code'

git push origin; git push origin --tags

```

4. **Instalar qemu**

```bash

sudo apt-get install qemu-system-riscv64 gcc-riscv64-linux-gnu

```

5. **Compilar e iniciar xv6**: posicionarse en el directorio so24lab3gXX (debe haber un archivo “Makefile”) y ejecutar el sistema usando qemu.

Como en este laboratorio haremos hincapié en mediciones de rendimiento,

Importante: Aunque `xv6-riscv` soporta múltiples procesadores, debemos ejecutar nuestras mediciones lanzando la máquina virtual con un único procesador utilizando

```bash

make CPUS=1 qemu

```

Objetivos

El planificador apropiativo de `xv6-riscv` utiliza un algoritmo sencillo para distribuir tiempo de procesador entre los procesos en ejecución, pero esto tiene un costo aparejado. Los objetivos de este laboratorio son:

- **Estudiar** el funcionamiento del scheduler original de xv6-riscv
- **Analizar** los procesos que se benefician/perjudican con esta decisión de diseño
- **Desarrollar** una implementación reemplazando la política de planificación por una propia que deberá respetar ciertas condiciones
- **Analizar** cómo la nueva política afecta a los procesos en comparación con el planificador original.
```

Entrega

- Deberán **ENTREGAR UN INFORME** con todo el análisis que realicen.
    - Debe estar en formato Markdown
    - Debe estar incluido en el repositorio del grupo con el nombre INFORME.md
    - Agreguen el informe al repositorio al principio y vayan haciendo commits también de su contenido. De esta forma podemos ver qué partes escribió cada uno.
- Deberán entregar el código de su implementación en el repositorio del grupo para este laboratorio en bitbucket, con un directorio `xv6-riscv` dentro sobre el cual deberán hacer sus modificaciones. **No copiar del laboratorio anterior**, comenzar con una copia limpia de `xv6-riscv`.
    - Debe tener un commit inicial con el código limpio que descargaron de xv6 sin ninguna modificación.
    - Debe tener un branch main o master con el código que utilizan para analizar y medir el rendimiento del planificador original
    - Debe tener un branch llamado mlfq con el código que utilizan para analizar y medir el rendimiento del nuevo planificador mlfq.
- Deberán respetar el *coding style* de xv6. A rajatabla. No hay negociaciones ni excepciones.
- Deberán entregar la versión final utilizando los mismos criterios que en lab 2 antes de las **23:59 del 31 de octubre**.

Implementación

Primera Parte: Estudiando el planificador de xv6-riscv

Comenzaremos este laboratorio leyendo código para entender cómo funciona la planificación en `xv6-riscv`:

Analizar el código del planificador y responda en el informe:

1. ¿Qué política de planificación utiliza `xv6-riscv` para elegir el próximo proceso a ejecutarse?
2. ¿Cúales son los estados en los que un proceso puede permanecer en xv6-riscv y qué los hace cambiar de estado?
3. ¿Qué es un *quantum*? ¿Dónde se define en el código? ¿Cuánto dura un *quantum* en `xv6-riscv`?
   Pista: Se puede empezar a buscar desde la system call `uptime` o leyendo la documentación de xv6 en la sección de interrupciones.
4. ¿En qué parte del código ocurre el cambio de contexto en `xv6-riscv`? ¿En qué funciones un proceso deja de ser ejecutado? ¿En qué funciones se elige el nuevo proceso a ejecutar?
5. ¿El cambio de contexto consume tiempo de un *quantum*?

Segunda Parte: Medir operaciones de cómputo y de entrada/salida

Para ver cómo el planificador de `xv6-riscv` afecta a los distintos tipos de procesos en la práctica, deberán integrar a `xv6-riscv` los programas de espacio de usuario `iobench` y `cpubench` (que adjuntamos en el aula virtual). Estos programas realizan mediciones de operaciones de escritura/lectura y operaciones de cómputo, respectivamente.
Las funciones son:

* iobench(int N): Ejecuta N veces un experimento donde cuenta operaciones de entrada/salida. Cada experimento abre un archivo para escritura y luego escribe el contenido de un buffer, y repite las operaciones para lectura, es decir, operaciones de entrada/salida.

* cpubench(int N): Ejecuta N veces un experimento donde cuenta operaciones de cómputo. Cada ciclo de medición realiza varias multiplicaciones de matrices MxM, es decir, operaciones de cómputo intensivo.

Experimento 1: ¿Cómo son planificados los programas iobound y cpubound?

En esta sección van a tener que completar los programas iobench y cpubench con una *métrica* que les permita comparar la cantidad de operaciones de cada tipo que se realizan a medida que cambiemos los parámetros del sistema operativo.

En cada ciclo de medición las operaciones (de input/output o de cómputo) se ejecutarán muchas veces. El largo del experimento (o sea, cuántas veces se ejecutan las operaciones) y el tamaño de las operaciones pueden cambiarlo en base a los siguientes criterios:
1. Que el experimento sea lo suficientemente largo como para que haya cambios de contexto entre los procesos.
2. Que el experimento sea lo suficientemente corto como para que el programa entero de mediciones se ejecute en un tiempo razonable, digamos 1 minuto.
3. Que compile: si las operaciones son demasiado grandes, podemos quedarnos sin memoria en el al ejecutar muchos procesos iobench o que el conteo de la cantidad de operaciones de cpu de overflow.
4. Que las mediciones no sean menores ni cercanas a 1 en la mayoría de los casos, ya que son casteadas a enteros y se pierde mucha información en el redondeo.

Utilizando el largo de quantum **10 veces más pequeño que el original** y un valor de N constante, medir cuántas veces son planificados los procesos en los siguientes escenarios:

a. `iobench N &`
b. `iobench N &; iobench N &; iobench N &`
c. `cpubench N &`
d. `cpubench N &; cpubench N &; cpubench N &`
e. `iobench N &; cpubench N &; cpubench N &; cpubench N &`
f. `cpubench N &; iobench N &; iobench N &; iobench N &`

Otra forma de hacer estos experimentos es utilizando sólo la salida de uno de ellos. En este caso medimos la performance de un proceso en particular dado los otros procesos del entorno. Eso se puede realizar con un comando como

```
iobench 3 | iobench 3 | iobench 3 &
cpubench 3 | cpubench 3 | cpubench 3 &
```

Responder las siguientes preguntas **utilizando gráficos y/o tablas** para justificar sus respuestas:

1. Describa los parámetros de los programas cpubench e iobench para este experimento (o sea, los define al principio y el valor de N. Tener en cuenta que podrían cambiar en experimentos futuros, pero que si lo hacen los resultados ya no serán comparables).
2. ¿Los procesos se ejecutan en paralelo? ¿En promedio, qué proceso o procesos se ejecutan primero? Hacer una observación cualitativa.
3. ¿Cambia el rendimiento de los procesos iobound con respecto a la cantidad y tipo de procesos que se estén ejecutando en paralelo? ¿Por qué?
4. ¿Cambia el rendimiento de los procesos cpubound con respecto a la cantidad y tipo de procesos que se estén ejecutando en paralelo? ¿Por qué?
5. ¿Es adecuado comparar la cantidad de operaciones de cpu con la cantidad de operaciones iobound?

Experimento 2: ¿Qué sucede cuando cambiamos el largo del quantum?

En esta sección, deben achicar el largo del quantum primero a 10000 y luego a 1000, y volver a repetir los mismos experimentos. Es posible que necesiten cambiar también sus métricas de medición para que los resultados sean comparables, pero no deberían cambiar los parámetros del experimento (o sea, N, IO_OPSIZE, IO_EXPERIMENT_LEN, etc.) Luego responder:

1. ¿Fue necesario modificar las métricas para que los resultados fueran comparables? ¿Por qué?

2. ¿Qué cambios se observan con respecto al experimento anterior? ¿Qué comportamientos se mantienen iguales?
3. ¿Con un quantum más pequeño, se ven beneficiados los procesos iobound o los procesos cpubound?

Tercera Parte: Asignar prioridad a los procesos

Para esta parte deberán crear una rama en su repositorio con nombre `mlfq`.

Habiendo visto las propiedades del planificador existente, lo vamos a reemplazar con un planificador MLFQ de tres niveles. Esto lo deben hacer de manera gradual. El primer paso será mantener un registro de la prioridad de los procesos, sin que esto afecte la planificación.

1. Agregue un campo en `struct proc` que guarde la prioridad del proceso (entre `0` y `NPRIO-1` para `#define NPRIO 3` niveles en total siendo `0` el prioridad minima y el `NPRIO-1` prioridad máxima) y manténgala actualizada según el comportamiento del proceso, además agregue el campo en `struct proc` que guarde la cantidad de veces que fue elegido ese proceso por el planificador para ejecutarse y se mantenga actualizado:
    - **MLFQ regla 3**: Cuando un proceso se inicia, su prioridad será máxima.
    - **MLFQ regla 4**: Descender de prioridad cada vez que el proceso pasa todo un *quantum* realizando cómputo. Ascender de prioridad cada vez que el proceso se bloquea antes de terminar su *quantum*. **Nota**: Este comportamiento es distinto al del MLFQ del libro.
2. Para comprobar que estos cambios se hicieron correctamente, modifique la función `procdump` (que se invoca con `CTRL-P`) para que imprima la prioridad de los procesos. Así, al correr nuevamente `iobench` y `cpubench`, debería darse que luego de un tiempo que los procesos `cpubench` tengan baja prioridad mientras que los `iobench` tengan alta prioridad.

Cuarta Parte: Implementar MLFQ

Finalmente implementar la planificación propiamente dicha para que nuestro `xv6-riscv` utilice MLFQ.

1. Modifique el planificador de manera que seleccione el próximo proceso a planificar siguiendo las siguientes reglas:
    - **MLFQ regla 1**: Si el proceso A tiene mayor prioridad que el proceso B, corre A. (y no B)
    - **MLFQ regla 2**: Si dos procesos A y B tienen la misma prioridad, corre el que menos veces fue elegido por el planificador.

2. Repita las mediciones de la segunda parte para ver las propiedades del nuevo planificador.
3. Para análisis responda: ¿Se puede producir *starvation* en el nuevo planificador? Justifique su respuesta.

Importante:

- Mucho cuidado con el uso correcto del mutex de cada proceso.
- Los experimentos deben ser realizados en entornos similares: computadora, sin otras tabs o procesos abiertos, etc.
- Los experimentos deben ser realizados con los mismos parámetros como
    - N
    - #define CPU_MATRIX_SIZE 128
    - #define CPU_EXPERIMENT_LEN 256

Puntos estrella

Si quieren hacer puntos extras, cada uno debe estar implementado en una rama aparte del repositorio y documentado en el informe. **NO** deben estar implementados en la rama principal del repositorio.

- Del planificador:

1. Reemplace la política de ascenso de prioridad por la regla 5 de MLFQ de OSTEP: Priority boost.

2. Modifique el planificador de manera que los distintos niveles de prioridad tengan distintas longitudes de *quantum*.

3. Cuando no hay procesos para ejecutar, el planificador consume procesador de manera innecesaria haciendo *busy waiting*. Modifique el planificador de manera que ponga a dormir el procesador cuando no hay procesos para planificar, utilizando la instrucción `hlt`.

4. (Difícil) Cuando xv6-riscv corre en una máquina virtual con 2 procesadores, la performance de los procesos varía significativamente según cuántos procesos haya corriendo simultáneamente. ¿Se sigue dando este fenómeno si el planificador tiene en cuenta la localidad de los procesos e intenta mantenerlos en el mismo procesador?

5. Llevar cuenta de cuánto tiempo de procesador se le ha asignado a cada proceso, con una *system call* para leer esta información desde el espacio de usuario.