Sistemas Operativos - FaMAF - UNC

- Vesión 2021: Ignacio Moretti
- Versiones 2014, 2016-2020: Carlos Bederián
- Versión 2015: Facundo Ramallo, Pablo Ventura

Objetivos

El planificador apropiativo de `xv6` 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; **analizar** los procesos que se benefician/perjudican con esta decisión de diseño; por último **desarrollar** una implementación que mejore esta situación reemplazando la política de planificación por una propia que deberá respetar ciertos condiciones.

Primera Parte: Estudiando el planificador de `xv6`

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

1. Analizar el código del planificador y responda: ¿Qué política utiliza `xv6` para elegir el próximo proceso a correr? Pista: `xv6` nunca sale de la función `scheduler` por medios "normales".

2. Analizar el código que interrumpe a un proceso al final de su *quantum* y responda:

   a. ¿Cuánto dura un *quantum* en `xv6`?

   b. ¿Hay alguna forma de que a un proceso se le asigne menos tiempo?

   Pista: Se puede empezar a buscar desde la system call `uptime`.

Segunda Parte: Cómo el planificador afecta a los procesos

Pasamos a ver cómo el planificador de `xv6` afecta a los distintos tipos de procesos en la práctica. Para ello se deberán integrar a `xv6` los programas de espacio de usuario `iobench` y `cpubench` (que adjuntamos en el aula virtual). Estos programas realizan mediciones (no muy precisas) de respuesta de entrada/salida y de poder de cómputo, respectivamente.

Importante: Aunque `xv6` soporta múltiples procesadores, debemos ejecutar nuestras mediciones(`iobench` y `cpubench`) lanzando la máquina virtual con un único procesador. (i.e. `make CPUS=1 qemu-nox`)

1. Mida la respuesta de I/O y el poder de cómputo obtenido para las distintas combinaciones posibles entre 0 y 2 `iobench` junto con entre 0 y 2 `cpubench`, y grafique los resultados en el informe.

   Caso 0: 1 `iobench`

   Caso 1: 1 `iobench` 1 `cpubench`

   Caso 2: 1 `iobench` 2 `cpubench`

   Caso 3: 1 `cpubench`

   Caso 4: 1 `cpubench` 2 `iobench`

   Caso 5: 2 `cpubench` 2 `iobench`

   Caso 6: 2 `cpubench`

   Caso 7: 2 `iobench`

2. Repita el experimento para *quantums* 10, 100 y 1000 veces más cortos. Tenga en cuenta que modificar el *tick* afecta el funcionamiento de `iobench` y `cpubench`, o sea que quizás necesite modificarlos para que mantengan un funcionamiento similar para que se puedan comparar los resultados en los distintos escenarios de prueba.

   Escenario 0: *quantum* por defecto, se corren los casos anteriores (caso 0 al 7, no lo tienen que repetir usen los resultados del apartado anterior)

   Escenario 1: *quantum* 10 veces más corto, se corren los casos anteriores (caso 0 al 7)

   Escenario 2: *quantum* 100 veces más corto, se corren los casos anteriores (caso 0 al 7)

   Escenario 3: *quantum* 1000 veces más corto, se corren los casos anteriores (caso 0 al 7)

Tercera Parte: Rastreando la prioridad de los procesos

Habiendo visto las propiedades del planificador existente, lo reemplazar con un planificador MLFQ de tres niveles. A esto se debe hacer de manera gradual, primero rastrear 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) y manténgala actualizada según el comportamiento del proceso:

   - 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 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 `cpubench` tenga baja prioridad mientras que `iobench` tenga alta prioridad.

Cuarta Parte: Implementando MLFQ

Finalmente implementar la planificación propiamente dicha para que nuestro `xv6` 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, corren en *round-robin* por el *quantum* determinado.
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 `ptable.lock`.

Extras

- 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 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. (Muy difícil) Y si no quisiéramos usar los ticks periódicos del timer por el problema de (1), ¿qué haríamos? Investigue cómo funciona e implemente un tickless kernel.
- De las herramientas de medición:
    - Llevar cuenta de cuánto tiempo de procesador se le ha asignado a cada proceso, con una *system call* para leer esta información desde espacio de usuario.

Entrega

- Deberán entregar via commits+push al repositorio del grupo para este laboratorio en bitbucket, con un directorio `xv6` dentro sobre el cual deberán hacer sus modificaciones. No copiar laboratorios anteriores, comenzar en limpio.
- El *coding style* deberá respetar a rajatabla las convenciones de `xv6`.
- Tienen 3 semanas para realizar el laboratorio, fecha de entrega hasta el **Martes 25/10/2021 23:59h**.