CC4302
Sistemas Operativos
Profesor: Luis Mateu

- Núcleo clásico vs. núcleo moderno
- Núcleo monolítico vs. micro núcleo
- El scheduler de Linux: prioridad estática, dinámica, colas de activos y expirados
- Modelos de consistencia de memoria
- Consistent locking behavior

Núcleo clásico vs. núcleo moderno

Clásico	***Moderno***
fines 70: System V 3.x, BSD, Linux 1, Win. 95	años 90: System V 4.x, Solaris 2, Linux 2, Win. NT
Simple	Complejo
Liviano en uso de CPU y memoria	Consumidor de CPU y memoria
Secuencial	Paralelo
Ideal para mono-core	Ideal para multi-core
Un solo core activo en el núcleo con interrupciones inhibidas	Múltiples threads en paralelo con interrupciones permitidas
Un spin-lock controla la entrada al núcleo	Múltiples spin-locks: 1 por cada estructura crítica
No hay dataraces ni deadlocks	Errores de programación pueden llevar a dataraces y deadlocks
Lento para reaccionar a las interrupciones	Rápido para reaccionar a interrupciones: *tiempo real*

Núcleo monolítico vs. micro núcleo

Núcleo monolítico	***micro núcleo***
Muchas funciones en el núcleo	Solo funciones esenciales: procesos y memoria
Muchos servicios brindados por procesos	Casi todo los servicios brindados por procesos
Sistema de archivos, la red, drivers en el núcleo	El sistema de archivos, la red, drivers en procesos
Sistema gráfico en un proceso (Ej.: X-windows)	%
Complejo	Simple
Inseguro	Seguro
Difícil lograr estabilidad	Simple de lograr estabilidad
Eficiente en cambios de contexto	Alto sobrecosto en cambios de contexto

El scheduler de Linux

- Es interesante porque reune características de round robin, prioridades y shortest job first
- Implementa tajadas de tiempo (*time slicing*)
- Implementa prioridades estáticas y dinámicas
- Le da prioridad a los procesos con ráfagas cortas como los procesos intensivos en entrada/salida o procesos interactivos
- Cumple con los requisitos de tiempo real: todos los cálculos se realizan en un tiempo acotado por una constante
- Le da prioridad a los procesos establecidos como de tiempo real, pero requiere privilegio de administrador (*root*)

Prioridad estática

- La *prioridad estática* va entre -20, la mejor prioridad, hasta 19, la peor
- 0 es la prioridad estándar: la del shell
- Se establece con el comando *nice(1)*:

  nice -n*[=pri] comando* ...
- Ejecuta el comando sumándole *pri* puntos a las prioridad del shell de comandos, lo que se traduce en peor prioridad
- O también la llamada al sistema *nice(2)*
- Los usuarios normales solo pueden empeorar la prioridad de sus procesos
- Solo *root* puede mejorar la prioridad de un proceso usando la llamada a sistema *nice(2)*
- *La prioridad estática solo se usa para calcular la duración de la tajada de tiempo como (20 – prioridad estática) x 10*
- Los procesos con mejor prioridad estática tienen tajadas más extensas

Prioridad dinámica

Se calcula de la siguiente manera:

- Cada vez que un proceso completa una ráfaga, se resta la duración de esa ráfaga a lo que le queda de tajada y también a su *tiempo dormido*
- Cada vez que un proceso pasa a estado READY, se suma la duración de su estado de espera a su *tiempo dormido*
- El *tiempo dormido* no puede bajar de 0 segundos o exceder un segundo
- Los procesos intensivos en entrada salida tienen un tiempo dormido cercano a 1 segundo, mientras que los intensivos en CPU se acercan a 0
- Cuando un proceso pasa a estado READY se le entrega un *bono* calculado como $\lfloor tiempo\ dormido/100 \rfloor$ lo que da un valor entre 0 y 10
- La *prioridad dinámica* es: *prioridad estática*+21-*bono*+5
- No puede bajar de 1 ni exceder 40
- *El scheduler siempre elige ejecutar el proceso con la prioridad dinámica de menor valor*

Cola de activos y cola de expirados

- El scheduler solo elige procesos de la *cola de activos*
- Cuando un proceso agota su tajada, pasa a la *cola de expirados* y ya no es elegible por el scheduler por un rato
- Cuando el scheduler se encuentra con la *cola de activos* vacía, intercambia la *cola de expirados* por la *cola de activos* con lo que los procesos que habían completado su tajada de tiempo vuelve a ser elegibles por el scheduler
- La prioridad dinámica de un proceso se calcula una sola vez: cuando pasa a estado READY
- Cada nivel de prioridad dinámica tiene su propia cola FIFO
- Agregar o extraer procesos toma tiempo constante
- El scheduler mantiene una máscara de bits que indica las colas que no están vacías
- Determinar cuál es la cola no vacía con mejor prioridad toma tiempo constante
- Los procesos de tiempo real tienen prioridad inferior a 1

El scheduler de Linux combina balanceadamente las ventajas de round robin, prioridades y *shortest job first*

Problema

- 2 threads ejecutan cada uno:

```
volatile int flag1=0, flag2=0, v1=0, v2=0;
int z, w;
```

```
v1= 100;
flag1= 1
while (!flag2)
  ;
z= v2;
```

```
v2= 200;
flag2= 1
while (!flag1)
  ;
w= v1;
```

- Al terminar se espera que *z* sea 200 y *w* sea 100
- Así será si el *modelo de consistencia de memoria* es secuencial
- Pero no si es *relajado*
- El primer thread puede ejecutar v1=100 seguido de flag1=1
- Pero el segundo thread podría ver que primero ocurrió flag1=1 y luego v1=100
- ¡En tal caso *w* podría quedar en 0!
- *Se evita usando un mutex o un semáforo*

Modelos de consistencia de memoria

- Son las reglas de ordenamiento de los accesos a la memoria en un computador multi-core
- En el *modelo de consistencia de memoria* secuencial todos los cores ven un mismo orden en los accesos a memoria
- Simple de implementar cuando todos los cores comparten el mismo bus para acceder a la memoria
- Pero en computadores modernos, para mayor eficiencia, cada core tiene sus propios canales de memoria
- La mayoría de los accesos de un core van a sus propios canales de memoria
- Líneas de comunicación entre cores permiten acceder a la memoria de otros cores, transparentemente, como si fuesen memoria propia, aunque no tan rápidamente: *Non Uniforn Memory Access* (NUMA)
- Los cores pueden ver órdenes de acceso diferente a la memoria, lo que abre el camino a los *modelos de consistencia de memoria relajados*

Consistent Locking Behavior

- Para evitar los *dataraces*, un programador debe usar consistentemente mutex para ordenar los accesos a la memoria
- Significa que cada vez que dos cores acceden a una misma dirección de memoria *d*, con uno modificando esa dirección, debe existir un mutex *m* tal que un core solicitó el mutex *m* antes del acceso a *d*, lo liberó después del acceso, el segundo mutex solicitó *m*, accedió a *d* y finalmente liberó *m*, en ese orden
- La sincronización con el mutex usa la instrucción *swap* que es una *barrera de memoria*: si un core hizo un acceso a memoria antes del *swap*, el resto de los cores también deben ver ese acceso ocurrir antes del *swap*
- Lo mismo con un acceso que ocurrió después del *swap*
- *Sanitize* para threads verifica que se cumple el *consistent locking behavior*
- Si detecta un problema, es altamente probable que tenga un datarace
- Problema: *falsos positivos*

Conclusiones

- La programación multi-core es más complicada de lo que parece
- Evítela a menos que realmente tenga problemas de eficiencia
- Si recurre a ella, verifique sus programas con sanitize para threads
- Sea consistente en usar un mutex para ordenar los accesos a la memoria compartida
- Si es posible, recurra a procesos que no comparten la memoria