Tema 6: Implementación

Elvira Albert

Departamento de Sistemas Informáticos y Computación

Universidad Complutense de Madrid

elvira@sip.ucm.es

Madrid, Abril, 2021

Introducción

Motivación

- En los temas anteriores hemos visto mecanismos de programación concurrente y técnicas para usarlos
- Los multi-procesadores proporcionan normalmente instrucciones máquina para implementar locks y barreras
- Los mecanismos de programación concurrente (procesos, semáforos, monitores) se implementan en el software

Kernel

- Núcleo de cualquier programa concurrente
- Tiene el role de proporcionar un procesador virtual a cada proceso de manera que tienen la ilusión de ejecutar en su propio procesador
- Tiene un conjunto pequeño de estructuras de datos (representan estado de los procesos, semáforos, etc.)
- Subrutinas para implementar las primitivas de concurrencia

Contenidos Tema 6

6.1 Kernel en único procesador

Instrucción co generaliza a process y se implementa:

```
S0;
co P1:S1;
// ...
// Pn:Sn;
oc
Sn+1
```

```
S0;
for[i=1 to n]
   fork(Pi);
for[i=1 to n]
   join(Pi);
```

1. *fork*: creación de un proceso child (como argumento recibe la dirección de las instrucciones a ejecutar)
2. *quit*: el proceso deja de existir
3. *join*: espera a que termine un proceso hijo

6.1 Kernel en único procesador

Kernel que implementa fork, join y quit:

- Las estructuras de datos representan los procesos
- 3 tipos de subrutinas
  - **manejadores de interrupción**: atienden interrupciones de los procesos (cuando encuentran primitivas de concurrencia) e interrupciones externas (de periféricos, etc.)
  - **primitivas de concurrencia**: en un procesador único se ejecutan atómicamente, en multi-procesadores los procesos pueden ejecutar estas primitivas concurrentemente
  - **dispatcher**: queremos garantizar fairness, para ello se utiliza un timer proporcionado por el hardware que permite periódicamente pasar el control de un proceso a otro
- Asumimos que siempre hay un proceso disponible, que inicialmente hay uno ejecutando

6.1 Kernel en único procesador

```
processType processDescriptor[maxProcs];
int executing=0;
lists free, ready, waiting;
SVC_handler:
  { guardar estado executing;
    determinar primitiva invocada y llamarla;}
Time_handler:
  { insertar descriptor executing en ready;
    executing=0; dispatcher();}
procedure fork(estado inicial)
  { sacar un descriptor de free e insertarlo en ready;
    dispatcher();
  }
```

6.1 Kernel en único procesador

```
procedure quit()
  { guardar executing terminado, insertar descriptor free;
    executing=0;
    if (parent waiting)
      {sacar parent de waiting e insertarlo en ready;}
    dispatcher();}
procedure join(nombre hijo)
  { if (hijo no terminado)
      {insertar descriptor executing en waiting;
       executing = 0; }
    dispatcher();}
procedure dispatcher()
 { if (executing==0)
      { sacar descriptor de ready;
        poner executing con indice de descriptor; }
  start timer;
  cargar estado de executing;}
```

6.2 Kernel multi-procesador

Cambios principales:

- Almacenar los procesos y estructuras de datos en memoria compartida
- Acceder a las estructuras de datos en exclusión mutua
- El dispatcher debe explotar los múltiples procesadores

Asumimos:

- todo procesador tiene un timer
- las interrupciones las recibe el mismo procesador que estaba ejecutando el proceso que lanzó la interrupción
- cuando un procesador está idle, examina la cola ready, y si está vacía comienza a ejecutar el proceso Idle
- está ejecutando el procesador i
- se utilizan los algoritmos del tema 3 para lock y unlock

6.2 Kernel multi-procesador

```
processType processDescriptor[maxProcs];
int executing[max];
lists free, ready, waiting;
SVC_handler:
  { guardar estado executing[i];
    determinar primitiva invocada y llamarla;}
Time_handler:
  { lock ready; insertar descriptor executing[i] en ready;
    unlock ready;
    executing[i]=0; dispatcher();}
procedure fork(estado inicial)
  { lock free; sacar un descriptor de free; unlock free;
    lock ready; insertarlo en ready; unlock ready;
    dispatcher();
  }
```

6.2 Kernel multi-procesador

```
procedure quit()
  { guardar executing[i] terminado,
    lock free; insertar executing[i] free;
    unlock free;
    executing[i]=0;
    if (parent waiting)
      {lock waiting; sacar parent de waiting;
       unlock waiting;
       lock ready; insertarlo en ready; unlock ready;}
    dispatcher();}
procedure join(nombre hijo)
  { if (hijo terminado) return
    executing[i] = 0;
    lock waiting;
    insertar descriptor executing[i] en waiting;
    unlock waiting; dispatcher();}
```

6.2 Kernel multi-procesador

```
procedure dispatcher()
 { if (executing[i]==0)
      { lock ready;
        if (ready not empty)
            {sacar descriptor de ready;
             poner executing[i] con indice de descriptor;
            }
        else executing[i]=Idle
        unlock ready;
       }
  if (executing[i] not Idle) start timer;
  cargar estado de executing[i];
 }
```

6.2 Kernel multi-procesador

```
process Idle
{
  while (executing[i]==Idle)
    { while (ready is empty) Delay;
      lock ready;
      if (ready is not empty)
        { coger descriptor de ready;
          executing[i] = descriptor;
        }
      unlock ready;
    }
  start timer en procesador i;
  cargar estado de executing[i];
}
```

Fairness: no es suficiente con usar timer, se tiene que utilizar protocolos fair para acceder a las secciones críticas

6.3-6.4 Implementación semáforo y monitor en Kernel

Implementación semáforo en un Kernel

- Añadir semáforos al kernel con único procesador
- Añadir descriptores con los valores de los semáforos
- Tres primitivas: createSem, PSem, VSem
- Nuevo estado: bloqueado en semáforo
- Lista de procesos bloqueados en cada semáforo

Implementación monitores en un Kernel

- Cada monitor mName tiene un lock mLock y una cola de procesos esperando
- Los descriptores de procesos pueden estar en: ready, monitor_waiting, condition_waiting
- Primitivas: enter(monitor), exit(monitor), wait(monitor,cond), signal(monitor, cond)