CC4302
Sistemas Operativos
Profesor: Luis Mateu

Unidad 2: administración de procesos

- Implementación de secciones críticas para monocore
- Secciones críticas en un multicore
- Implementación de nSelf para multicore
- Implementación de FCFS para multicore
- Round Robin para multicore
- La rutina de atención del timer

Implementación de semáforos: nKernel/sem.c

```
int nSemWait(nSem *psem) {
   START_CRITICAL
   if (psem->count>0)
      psem->count--;
   else {
      nThread thisTh= nSelf();
      nth_putBack(psem->queue, thisTh);
      suspend(WAIT_SEM); // nth_fcfs1Suspend
      schedule();        // nth_fcfs1Schedule
   }
   END_CRITICAL
   return 0;
}
```

(*nth_scheduler.schedule)()

```
typedef struct {
  int count;
  void *queue;
} nSem;
```

Tipicamente en las herramientas de sincronización:

```
setReady(th);
suspend(WAIT…);
…
schedule();
```

```
int nSemPost(nSem *psem) {
   START_CRITICAL
   if (nth_emptyQueue(psem->queue))
      psem->count++;
   else {
      nThread w=
          nth_getFront(psem->queue);
      setReady(w); // nth_fcfs1SetReady
      schedule();    // nth_fcfs1Schedule
   }
   END_CRITICAL
   return 0;
}
```

Implementación de secciones críticas: caso single core

La única fuente de dataraces son las señales/interrupciones

START_CRITICAL (macro de C, invoca nth_startCritical)

- Inhibe las señales/interrupciones
- *nth_sigsetCritical* incluye: SIGALRM, SIGIO, SIGVTALRM,
- es equivalente a:

```
sigset_t sigsetOld;
pthread_sigmask(SIG_BLOCK, &nth_sigsetCritical, &sigsetOld);
```

END_CRITICAL (macro de C, invoca nth_endCritical)

- Permite nuevamente las señales/interrupciones
- es equivalente a:

```
sigset_t sigsetOld;
pthread_sigmask(SIG_SETMASK, &nth_sigsetApp, &sigsetold);
```

- En un núcleo real se inhiben las interrupciones con una instrucción de máquina como *disable* y se permiten nuevamente con la instrucción *enable*
- *Pero está prohibido su uso a nivel de usuario*

Secciones críticas en un multicore

- Además de las señales, los dataraces se pueden producir por no respetar la exclusion mutua de múltiples cores en una sección crítica
- START_CRITICAL es equivalente a:

```
sigset_t sigsetOld;
pthread_sigmask(SIG_BLOCK, &nth_sigsetCritical, &sigsetOld);
llLock(&nth_schedMutex); // pthread_mutex_lock
```

- nth_schedMutex es un mutex de tipo LLMutex que es el tipo pthread_mutex_t nativo de pthreads
- Asegura la exclusion mutua de los cores
- Si nth_schedMutex está ocupado, el core se bloquea
- No confundir con el tipo nMutex que corresponde a los mutex virtualizados de nThreads, es decir aseguran la exclusion mutua de los nthreads
- Si un nMutex está ocupado, el nthread se bloquea, pero el core puede y debe ejecutar otros nthreads
- END_CRITICAL es equivalente a:

```
llUnlock(&nth_schedMutex); // pthread_mutex_unlock
sigset_t sigsetOld;
pthread_sigmask(SIG_SETMASK, &nth_sigsetApp, &sigsetOld);
```

Implementación de nSelf() en multicore

- (En monocore basta una variable global)
- En un núcleo multicore nativo existen registros del procesador que guardan el número del core: 0, 1, 2, etc.
- En nThreads la función *nth_coreId()* sirve para obtener ese identificador
- Un arreglo almacena el thread en ejecución por cada core:
  nThread nth_coreThreads[ ]
- Con esto se implementa *nSelf* y *nth_setSelft*:

```
nThread nSelf() {  // The id of the running nthread
  return nth_coreThreads[nth_coreId()]; // ¡Datarace!
}

void nth_setSelf(nThread th) {
  nth_coreThreads[nth_coreId()]= th;
}
```

- ¿Cómo se implementa *nth_coreId*?
- Con *thread locals*: variables globales locales a un thread, cada thread tiene su propia instancia

```
__thread   int nth_thisCoreId;
#define nth_coreId() nth_thisCoreId
```

Scheduler FCFS para multicore: nKernel/sched-fcfs.c

- *La cola ready*

```
NthQueue *nth_fcfsReadyQueue;
```

- *Pasar un thread a estado READY*

```
void nth_fcfsSetReady(nThread th) {
    th->status= READY;
    if (nth_allocCoreId(th)<0)
        nth_putBack(nth_fcfsReadyQueue, th);
    else if (nth_allocCoreId(th)!=nth_coreId())
        nth_coreWakeUp(nth_allocCoreId(th));
}
```

- *Pasar un thread a estado de espera*

```
void nth_fcfsSuspend(State waitState) {
    nThread th= nSelf();
    th->status= waitState;
}
```

Caso multicore

- Puede ocurrir que un thread esté en estado de espera pero aún así estar asignado a un core, porque hay abundancia de cores
- El core está esperando en un sigsuspend a que algún thread pase a estado READY
- Cuando eso ocurre *setReady* invoca a nth_coreWakeUp para despertar al core asignado
- Esta estrategia no es estrictamente FCFS

- Puede ocurrir que varios threads pasen a estado READY y luego se invoque el scheduler
- Si hay cores disponibles, el scheduler se encarga de despertarlo para que ejecuten los threads agregados a la cola (función *nth_reviewCore*)

Scheduler FCFS para multicore: nKernel/sched-fcfs.c

```
void nth_fcfsSchedule(void) {
   nThread thisTh= nSelf();
   for ( ; ; ) {
      if ( thisTh!=NULL &&
           (thisTh->status==READY || thisTh->status==RUN) ) {
         break;
      nThread nextTh= nth_getFront(nth_fcfsReadyQueue);
      if (nextTh!=NULL) {
         nth_changeContext(thisTh, nextTh); // _changeToStack
         nth_setSelf(thisTh); // Set current running thread
         if (thisTh->status==READY)
            break;
      }
      nth_coreIsIdle[nth_coreId()]= 1; // To prevent recursive calls
      llUnlock(&nth_schedMutex); // pthread_mutex_unlock
      sigsuspend(&nth_sigsetApp);
      llLock(&nth_schedMutex); // pthread_mutex_lock
      nth_coreIsIdle[nth_coreId()]= 0;
   }
   thisTh->status= RUN;
   nth_reviewCore(nth_peekFront(nth_fcfsReadyQueue));
}
```

Round Robin

- La variable global *nth_sliceNanos* indica el tamaño de la tajada de tiempo
- En el descriptor de proceso, el campo *sliceNanos* indica cuanto le queda de tajada a un thread que está READY o en estado de espera
- En el descriptor de proceso, el campo *startCoreNanos* indica a qué hora recibió el core un thread que está en estado RUN
- Cuando un thread *th* pasa a estado de espera, descuenta de *th->slicesNanos* lo que ocupó de su tajada
- Si el scheduler descubre que *th->sliceNanos* es <= 0, le otorga la tajada completa, pero lo envía al final de la cola
- Por simplicidad, un thread que pasa a estado READY y le queda tajada de CPU, se va al principio de la cola READY, pero no le quita la CPU al que está ejecutándose
- Solo se ejecuta de inmediato si tiene un core asignado

Round Robin

- Pasar a estado READY

```
static void nth_rrSetReady(nThread th) {
    th->status= READY;
    if (nth_allocCoreId(th)<0) { // No asignado a algún core
        if (th->sliceNanos>0) // Le queda tajada
            nth_putFront(nth_rrReadyQueue, th);
        else { // No le queda tajada
            th->sliceNanos= nth_sliceNanos;
            nth_putBack(nth_rrReadyQueue, th);
    }   }
    else if (nth_allocCoreId(th)!=nth_coreId())
         nth_coreWakeUp(nth_allocCoreId(th));
}
```

- Parar a estado de espera

```
void nth_fcfsSuspend(State waitState) {
    nThread th= nSelf();
    th->status= waitState;
}
```

Round Robin: el scheduler

```
void nth_rrSchedule(void) {
   nThread thisTh= nSelf();
   if (thisTh!=NULL) {
      long long endNanos= nth_getCoreNanos();
      thisTh->sliceNanos -= endNanos-thisTh->startCoreNanos;
   }
   for (;;) {
      if (thisTh!=NULL
          && (thisTh->status==READY || thisTh->status==RUN) ) {
         if (thisTh->sliceNanos>0)
            break; // Continue running same allocated thread
         else {      // No slice remaining
            thisTh->sliceNanos= nth_sliceNanos;
            thisTh->status= READY;
            nth_putBack(nth_rrReadyQueue, thisTh);
         }
      }
```

Round Robin: el scheduler

```
        nThread nextTh= nth_getFront(nth_rrReadyQueue);
        if (nextTh!=NULL) {
            nth_changeContext(thisTh, nextTh);
            nth_setSelf(thisTh);
            if (thisTh->status==READY)
                break;
        }
        nth_coreIsIdle[nth_coreId()]= 1;
        llUnlock(&nth_schedMutex);
        sigsuspend(&nth_sigsetApp);
        llLock(&nth_schedMutex);
        nth_coreIsIdle[nth_coreId()]= 0;
    }
    thisTh->status= RUN;
    thisTh->startCoreNanos= nth_getCoreNanos();
    nth_reviewCores(nth_peekFront(nth_rrReadyQueue));
    nth_setCoreTimerAlarm(thisTh->sliceNanos, nth_coreId());
}
```

La rutina de atención de SIGVTALRM

```
static void VTimerHandler(int sig, siginfo_t *si, void *uc)
{
   if (nth_coreIsIdle[nth_coreId()])
      return; // prevent schedule recursive
   nThread th= nSelf();
   if (th==NULL)
      return; // to avoid weird race conditions
   START_HANDLER // llLock(&nth_schedMutex);
   th->sliceNanos= 0; // end of slice
   if ( !nth_coreIsIdle[nth_coreId()] )
      schedule();  // give core to another thread
   END_HANDLER // llUnlock(&nth_schedMutex);
}
```

Conclusión

- Próxima clase: implementación de timeouts (sleep, pthread_cond_timedwait)
- Implementación de un núcleo nativo: son cores verdaderos, pero no hay pthread_mutex_t y pthread_cond_t para sincronizarlos
- Solución: spin-locks
- Son semáforos binarios de bajo nivel
- La espera se implementa con busy-waiting
- A partir de spin-locks se construyen LLMutex y LLCond
- Ejercicio: estudie la implementación de los mutex y condiciones de nThreads (nMutex y nCond) en el archivo nKernel/mutex-cond.c