CC4302
Sistemas Operativos
Profesor: Luis Mateu

Unidad 2: administración de procesos

- Implementación de timeouts: nSleepNanos
- Round robin (repaso)
- Implementación del scheduling round robin
- Implementación de la rutina de atención del timer de tiempo virtual

Implementación de nSleepNanos

```
int nSleepNanos(long long nanos) {
    START_CRITICAL

    nThread thisTh= nSelf();
    suspend(WAIT_SLEEP);
    nth_programTimer(nanos, NULL);
    schedule();

    END_CRITICAL
    return 0;
}
```

Implementación *nth_programTimer*

```
void nth_programTimer(long long nanos,
                      void (*wakeUpFun)(nThread th)) {
   nThread thisTh= nSelf();
   if (nanos>0) {
      long long currTime= nGetTimeNanos();
      long long wakeTime= currTime+nanos;
      if ( nth_emptyTimeQueue(nth_timeQueue) ||
           wakeTime-nth_nextTime(nth_timeQueue)<0 ) {
         // arm timer
         nth_setRealTimerAlarm(wakeTime-currTime);
      }
      thisTh->wakeUpFun= wakeUpFun;
      nth_putTimed(nth_timeQueue, thisTh, wakeTime);
   }
   else {
      setReady(thisTh);
   }
}
```

Manejo de timers

- Configuración

```
struct sigaction sigact;
sigact.sa_flags= SA_SIGINFO;
sigact.sa_sigaction= nth_RTimerHandler;
sigact.sa_mask= nth_sigsetCritical;
sigaction(SIGALRM, &sigact, NULL);

timer_t nth_realTimer;
struct sigevent sigev;
sigev.sigev_notify= SIGEV_SIGNAL;
sigev.sigev_signo= SIGALRM;
sigev.sigev_value.sival_ptr= &nth_realTimer;
timer_create(CLOCK_REALTIME, &sigev, &nth_realTimer);
```

- Programar timer

```
void nth_setRealTimerAlarm(long long nanos) {
  struct itimerspec spec;
  spec.it_value.tv_sec= nanos/1000000000;
  spec.it_value.tv_nsec= nanos%1000000000;
  spec.it_interval.tv_sec= 0;
  spec.it_interval.tv_nsec= 0;
  timer_settime(nth_realTimer, 0, &spec, NULL);
}
```

Rutina de atención del timer

```
void nth_RTimerHandler(int sig, siginfo_t *si, void *uc)
{
   START_HANDLER
   nth_wakeThreads();
   nThread th= nSelf();
   if (! nth_coreIsIdle[nth_coreId()] && th!=NULL)
      schedule();
   END_HANDLER
}
```

Implementación de nth_wakeThreads

```
void nth_wakeThreads(void) {
   long long currTime= nGetTimeNanos();
   // Wake up all threads with wake time <= currTime
   while ( !nth_emptyTimeQueue(nth_timeQueue) &&
              nth_nextTime(nth_timeQueue)<=currTime ) {
      nThread th= nth_getTimed(nth_timeQueue);
      if (th->wakeUpFun!=NULL)
         (*th->wakeUpFun)(th);
      setReady(th);
   }
   nth_setRealTimerAlarm(
         nth_emptyTimeQueue(nth_timeQueue) ?
         0 : nth_nextTime(nth_timeQueue)-currTime );
}
```

nth_cancelThread

```
void nth_cancelThread(nThread th) {
    nth_delTimed(nth_timeQueue, th);
    nth_wakeThreads(); // rearm timer if needed
}
```

Sirve en operaciones como pthread_cond_timedwait: el thread se active con signal o broadcast, el thread quedaría todavía con el timer armado. Se cancela invocando nth_cancelThread.

Se necesitará para la recepción de mensajes, con timeout de la clase auxiliar de mañana.

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: setReady y suspend

- 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));
}
```

- Pasar a estado de espera

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

Round Robin: el scheduler (pág. 1)

```
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 (pág. 2)

```
        nThread nextTh= nth_getFront(nth_rrReadyQueue);
        if (nextTh!=NULL) {
            nth_changeContext(thisTh, nextTh);
            break;
        }
        nth_corePark();  // sigsuspend
    }
    thisTh->status= RUN;
    thisTh->startCoreNanos= nth_getCoreNanos();
    if (!nth_emptyQueue(nth_rrReadyQueue))
        nth_reviewCores();
    nth_setCoreTimerAlarm(thisTh->sliceNanos, nth_coreId());
}
```

Manejo de timers de tiempo virtual

- Configuración

```
struct sigaction sigact;
sigact.sa_flags= SA_SIGINFO;
sigact.sa_sigaction= nth_VTimerHandler;
sigact.sa_mask= nth_sigsetCritical;
sigaction(SIGVTALRM, &sigact, NULL);
timer_t *nth_timerSet; // One for each virtual core
struct sigevent sigev;
sigev.sigev_notify= SIGEV_THREAD_ID;
sigev._sigev_un._tid= gettid();
sigev.sigev_signo= SIGVTALRM;
sigev.sigev_value.sival_ptr= &nth_timerSet[nth_coreId()];
timer_create(CLOCK_THREAD_CPUTIME_ID,
             &sigev, &nth_timerSet[nth_coreId()] );
```

- Programación:

```
void nth_setCoreTimerAlarm(long long sliceNanos, int coreId) {
  struct itimerspec slicespec;
  slicespec.it_value.tv_sec= sliceNanos/1000000000;
  slicespec.it_value.tv_nsec= sliceNanos%1000000000;
  slicespec.it_interval.tv_sec= 0;
  slicespec.it_interval.tv_nsec= 0;
  timer_settime(nth_timerSet[coreId], 0, &slicespec, NULL);
}
```

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

- 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 mutex y condiciones de bajo nivel
- Ejercicio: estudie la implementación de los mutex y condiciones de nThreads (nMutex y nCond) en el archivo nKernel/mutex-cond.c