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);
pthread_mutex_lock(&nth_schedMutex);
```

- 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:

```
pthread_mutex_unlock(&nth_schedMutex);
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[ ]
- Implementación incorrecta de *nSelf* y *nth_setSelft*:

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

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;
int nth_coreId(void) { return 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;
}
```

Scheduler FCFS 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)) {
         if (nth_allocCoreId(thisTh)<0 && thisTh->status==READY)
            nth_delQueue(nth_fcfsReadyQueue, thisTh);
         break;            // Continue running same allocated thread
      }
      nThread nextTh= nth_getFront(nth_fcfsReadyQueue);
      if (nextTh!=NULL) {
         nth_changeContext(thisTh, nextTh); // swapcontext
         break;
      }
      nth_corePark();
   }
   thisTh->status= RUN;
   if (!nth_emptyQueue(nth_fcfsReadyQueue))
      nth_reviewCores();
}
```

Estacionar un core (park)

```
int *nth_coreIsIdle;            // 0: working, 1: idle
sigset_t nth_sigsetCritical;    // Blocked in a critical section
sigset_t nth_sigsetApp;         // Accepted in app mode

void nth_corePark(void) {
   int coreId= nth_coreId();
   // To prevent a signal handler to call recursively this scheduler
   nth_coreIsIdle[coreId]= 1;
   if (nth_totalCores>1)
      nth_schedUnlock();
   sigsuspend(&nth_sigsetApp);
   if (nth_totalCores>1)
      nth_schedLock();
   nth_coreIsIdle[coreId]= 0;
}
```

Despertar un core

```
void nth_coreWakeUp(int id) {
   // send a signal to core id to wake it up from sigsuspend
   pthread_kill(nth_nativeCores[id], SIGUSR1);
}

void nth_reviewCores(nThread th) {
   if (th==NULL)
      return;
   int ncores= nth_totalCores;        // look for an idle core
   for (int id= 0; id<ncores; id++) {
      if (nth_coreIsIdle[id]) {
         if ( nth_coreThreads[id]==NULL ||
              nth_coreThreads[id]->status!=READY ) {
            // wake the core id up
            nth_coreWakeUp(id);
            break;
         }
      }
   }
}
```

Rutina de atención de SIGUSR1

- *Configuración*

```
struct sigaction sigact;
sigact.sa_flags= SA_SIGINFO;
sigact.sa_mask= nth_sigsetCritical;
sigact.sa_sigaction= nth_Usr1Handler;
sigaction(SIGUSR1, &sigact, NULL);
```

- *Rutina de atención*

```
static void nth_Usr1Handler(int sig, siginfo_t *si, void *uc) {
  // Do nothing
}
```