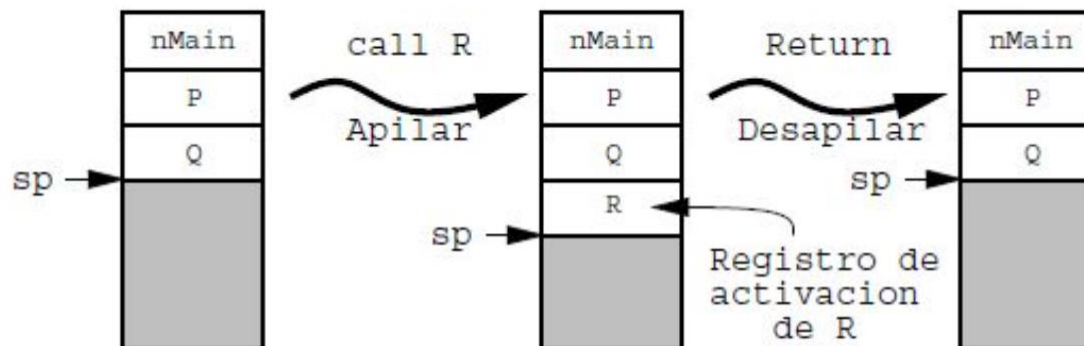
CC4302
Sistemas Operativos
Profesor: Luis Mateu

Unidad 2: administración de procesos

- Implementación de threads: La pila
- Threads nivel de núcleo (pthreads) vs. threads nivel usuario (nThreads)
- Compatibilidad de nThreads con pthreads
- El cambio de contexto
- Implementación de un scheduler FCFS para single core
- Implementación de semáforos
- Implementación de secciones críticas, caso single core

Implementación de threads: La pila

- Cada thread debe poseer su propia pila
- Ahí se apilan los registros de activación de las funciones pendientes: *frames*
- Eficiente: al llamar a una función, para crear su *frame* basta restar el tamaño al puntero a la pila
- Eficiente: al retornar, hay que sumar su tamaño al puntero a la pila
- Gracias al orden LIFO: la última función que se llamó será la primera en retornar, *considerando un solo thread*



Threads: Nivel núcleo vs nivel usuario

- Qué pasa con múltiples threads: *una pila por thread*
- Problema: ¿De qué tamaño?

Threads a nivel del núcleo: Ej. pthreads

- se colocan en un área de memoria dinámicamente extensible hasta 8 MB
- Desventaja: el espacio de direccionamiento es limitado, por lo que se puede crear un número limitado de pthreads

Threads a nivel usuario: Ej. nThreads

- No las implementa el núcleo
- Se implementan en base a funciones de biblioteca
- Las pilas solo se pueden colocar en el heap
- Si se crean de tamaño pequeño, se pueden crear muchos más threads: típicamente pilas de 16 KB
- Pero las pilas se desbordan con facilidad
- La detección del desborde es un problema
- Hay que modificar el compilador para que chequee el desborde

Alternativa que nadie implementa

- Modificar el compilador para que los frames se creen en el heap como cualquier estructura de datos
- Se eliminan las pilas
- No se desperdicia espacio para prevenir desborde de pila
- Ineficiente: puede llegar a ser 10x más lento que múltiples pilas
- Porque un programa gasta 10x o más en memoria para frames que en memoria para objetos en el heap
- Las pilas son pequeñas porque los *frames* son de corta vida
- La mejor solución hoy en día: threads a nivel del núcleo, como *pthreads*
- *nThreads* tiene propósitos pedagógicos: para mostrar cómo se administran los threads y procesos
- Virtualizar *n pthreads* para que ejecuten un número indeterminado de *nthreads* es un problema similar a virtualizar *n cores físicos* para que ejecuten un número indeterminado de *pthreads* (o procesos)

Compatibilidad de nThreads con pthreads

- En la implementación de nThreads, todas las funciones de la API tienen un nombre distinto al del estándar pthreads
- pthread_t es nThread
- pthread_create es nThreadCreate
- pthread_mutex_lock es nLock

  ... etc.

  Las funciones y variables globales que son parte de la implementación empiezan con ***nth_***

- Pero las aplicaciones deben incluir:

  #include "nthread.h"   // no "pthread.h"
- En ese archivo se traducen con macros los nombres estándares a los nombres de nThreads

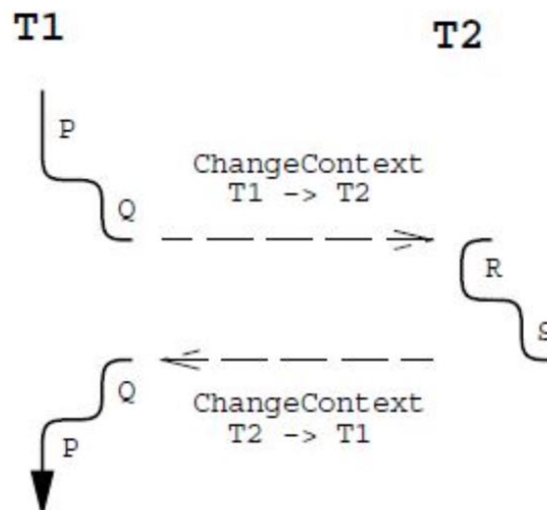
```
#define pthread_t nThread
#define pthread_create nThreadCreate
#define pthread_mutex_lock nLock
```

  ... etc.

- Solo algunas funciones de C y Unix se pueden invocar: aquellas en las que aparece una definición en ***nthread.h***

El cambio de contexto

- Considerando que el número de cores es limitado: un mismo core va ejecutando alternadamente distintos threads
- El cambio de contexto es cuando un core pasa de ejecutar un thread a otro
- *Restricciones*: el core tiene un solo juego de registros con un solo puntero a la pila (SP) y un solo puntero a la instrucción en ejecución (o contador de programa, PC)
- En el cambio de contexto se resguardan los registros del thread saliente y se restauran los del thread entrante



El cambio de contexto en nThreads: nKernel/nStack-amd64.s

- Es la única parte escrita en assembler
- No se puede programar en C
- El problema es similar a programar las funciones *setjmp/longjmp*, que también están escritas en assembler
- En nThreads*:* *_ChangeToStack(&spOut, &spIn);*
- Resguarda los registros en la pila del thread saliente con la pila apuntada por *spOut*
- Restaura los registros de la pila del thread entrante, con la pila apuntada por *spIn*
- También: *_CallInNewStack(&spOut, spNew, fun, ptr);*
- Se usa para llamar a la función raíz de un nuevo thread, con la pila apuntada por *spNew*
- Se restauran los registros en la pila del thread saliente, con la pila apuntada por *spOut*

Scheduler FCFS para single core: nKernel/sched-fcfs-1core.c

- *La cola ready*

```
NthQueue *nth_fcfs1ReadyQueue;
```

- *Pasar un thread a estado READY*

```
void nth_fcfs1SetReady(nThread th) {
    th->status= READY;
    nth_putBack(nth_fcfs1ReadyQueue, th);
}
```

- *Pasar un thread a estado de espera*

```
void nth_fcfs1Suspend(State waitState) {
    nThread th= nSelf();
    th->status= waitState;
}
```

Scheduler FCFS para single core: nKernel/sched-fcfs-1core.c

```
void nth_fcfs1Schedule(void) {
  nThread thisTh= nSelf();
  if ( thisTh!=NULL &&
      (thisTh->status==READY || thisTh->status==RUN)) {
    thisTh->status= RUN;
    return;
  }

  nThread nextTh= nth_getFront(nth_fcfs1ReadyQueue);
  while (nextTh==NULL) {
    nth_coreIsIdle[0]= 1; // To prevent recursive calls
    sigsuspend(&nth_sigsetApp);
    nth_coreIsIdle[0]= 0;
    nextTh= nth_getFront(nth_fcfs1ReadyQueue);
  }

  nth_changeContext(thisTh, nextTh); // _changeToStack

  nth_setSelf(thisTh); // Set current running thread
  thisTh->status= RUN;
}
```

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);
      schedule();
   }
   END_CRITICAL
   return 0;
}
```

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

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

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

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

END_CRITICAL (macro de C)

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

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

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