Sistemas Operativos

Universidad Complutense de Madrid

2016-2017

Práctica 3

Procesos e hilos: planificación y sincronización

J.C. Sáez

Contenido

Contenido

Introducción

Objetivos

- El objetivo principal de la práctica es implementar diferentes algoritmos de planificación en un entorno de simulación realista
  - El simulador proporcionado es un programa multi-hilo
  - Simula comportamiento del planificador del SO en un sistema con *n* CPUs
- Como objetivo instrumental, el alumno se familiarizará con el uso de POSIX Threads, semáforos, mutexes, variables condición

Introducción

Recuerda: Procesos vs. Hilos

- 2 procesos (padre - hijo) *no comparten nada* : se duplica toda la imagen de memoria
- 2 hilos (del mismo proceso) *comparten todo* menos la pila

Haced los ejercicios / ejemplos

- Ayudan a comprender la materia....
- ... y suelen acabar en preguntas del cuestionario

Contenido

Modo de uso

Terminal

```
debian:P3 osuser$ ./schedsim
No input file was provided
Usage: ./schedsim -i <input-file> [options]
debian:P3 osuser$ ./schedsim -h
List of options:
-h: Displays this help message
-n <cpus>: Sets number of CPUs for the simulator
-m <nsteps>: Sets the maximum number of simulation steps (default 50)
-s <scheduler>: Selects the scheduler for the simulation (default RR)
-d: Turns on debug mode
-p: Selects the preemptive version of SJF or PRIO (only if they are selected with -s)
-t <msecs>: Selects the tick delay for the simulator (default 250)
-q <quantum>: Set up the timeslice or quantum for the RR algorithm
-l <period>: Set up the load balancing period (specified in simulation steps)
-L: List available scheduling algorithms
debian:P3 osuser$ ./schedsim -L
Available schedulers:
RR
SJF
debian:P3 osuser$
```

Sintaxis de ficheros de tareas

Ejemplos proporcionados

- En la carpeta *examples* se incluyen varios ejemplos
- Es sencillo construir nuevos ejemplos siguiendo la sintaxis

Terminal

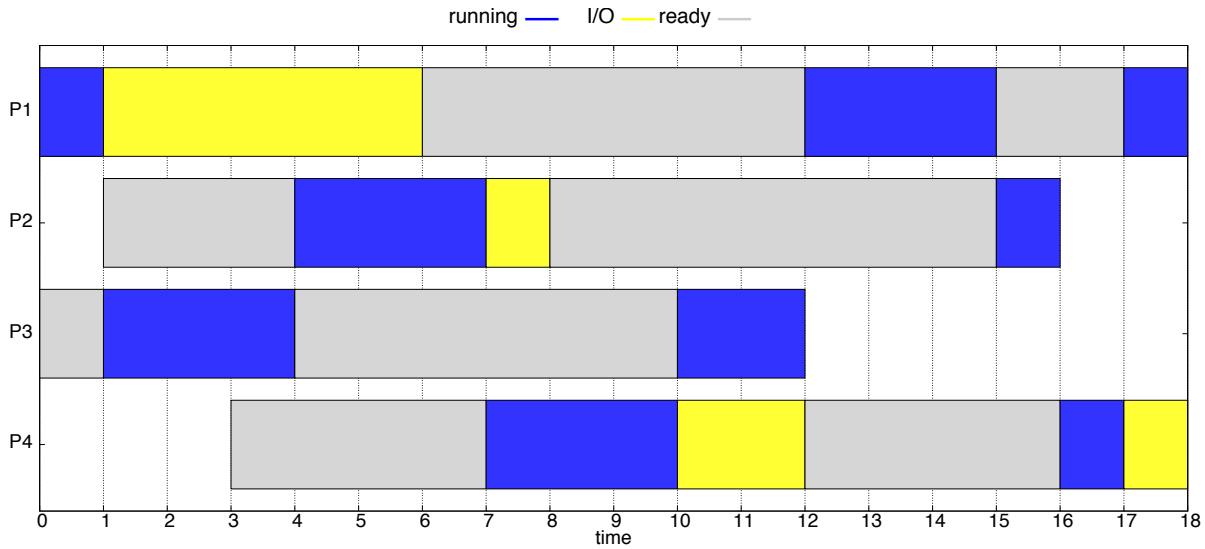
```
$ cat examples/example1.txt
P1 1 0 1 5 4
P2 1 1 3 1 1
P3 1 0 5
P4 1 3 3 2 1 1
```

- Una tarea por línea
  - Columna 1: nombre de la tarea
  - Columna 2: prioridad (*a menor valor* $\rightarrow$ *mayor prioridad*)
  - Columna 3: tiempo de llegada al sistema
  - Columnas siguientes: *ráfaga CPU - ráfaga E/S - ráfaga CPU - ...*

Ejemplo: RR con una CPU

Terminal

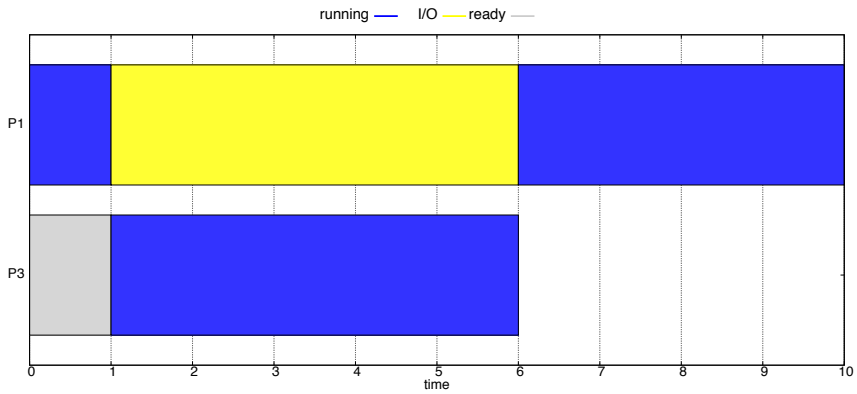
```
debian:P3 osuser$ ./schedsim -i examples/example1.txt
Statistics: task_name=P3 real_time=12 user_time=5 io_time=0
Statistics: task_name=P2 real_time=15 user_time=4 io_time=1
Statistics: task_name=P1 real_time=18 user_time=5 io_time=5
Statistics: task_name=P4 real_time=15 user_time=4 io_time=3
Simulation completed
Closing file descriptors...
debian:P3 osuser$ cd ../gantt-plot
debian:P3 osuser$ ./generate_gantt_chart ../schedsim/CPU_0.log
debian:P3 osuser$ cd -
debian:P3 osuser$ gnome-open CPU_0.eps
```

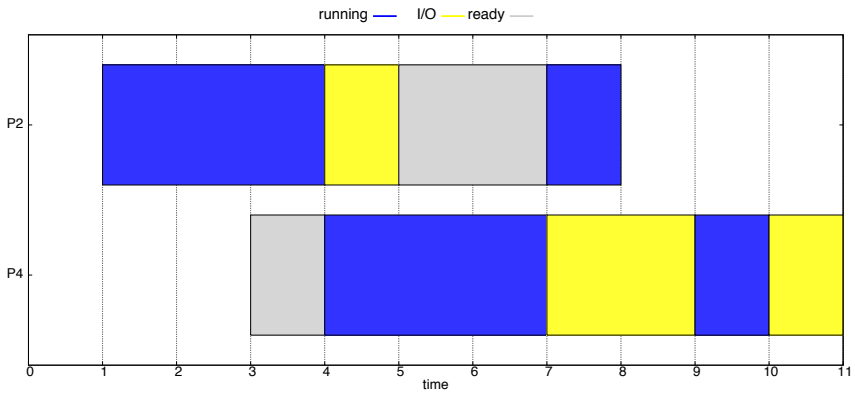


Ejemplo: RR con 2 CPUs

Terminal

```
debian:P3 osuser$ ./schedsim -i examples/example1.txt -n 2
Statistics: task_name=P3 real_time=6 user_time=5 io_time=0
Statistics: task_name=P2 real_time=7 user_time=4 io_time=1
Statistics: task_name=P1 real_time=10 user_time=5 io_time=5
Statistics: task_name=P4 real_time=8 user_time=4 io_time=3
Simulation completed
Closing file descriptors...
debian:P3 osuser$ cd ../gantt-plot
debian:P3 osuser$ ./generate_gantt_chart ../schedsim/CPU_0.log
debian:P3 osuser$ ./generate_gantt_chart ../schedsim/CPU_1.log
debian:P3 osuser$ cd -
debian:P3 osuser$ gnome-open CPU_0.eps
debian:P3 osuser$ gnome-open CPU_1.eps
```





Ejemplo: Modo depuración

- El modo depuración (opción -d) permite para visualizar qué ocurre cada ciclo de simulación
  - La opción "`-t <milisecs>`" permite establecer el tiempo de ciclo

Terminal

```
debian:P3 osuser$ ./schedsim -i examples/example1.txt -d -t 1000
==== TASK P1 ===
Priority: 1
Arrival time: 0
Profile: [ 1 5 4 ]
================
==== TASK P2 ===
Priority: 1
Arrival time: 1
Profile: [ 3 1 1 ]
...
Scheduler initialized. Press ENTER to start simulation.

CPU 0:(t0): New task P1
CPU 0:(t0): New task P3
CPU 0:(t0): P1 running
CPU 0:(t1): Task P1 goes to sleep until (t6)
CPU 0:(t1): New task P2
CPU 0:(t0): Context switch (P1)<->(P3)
CPU 0:(t1): P3 running
...
```

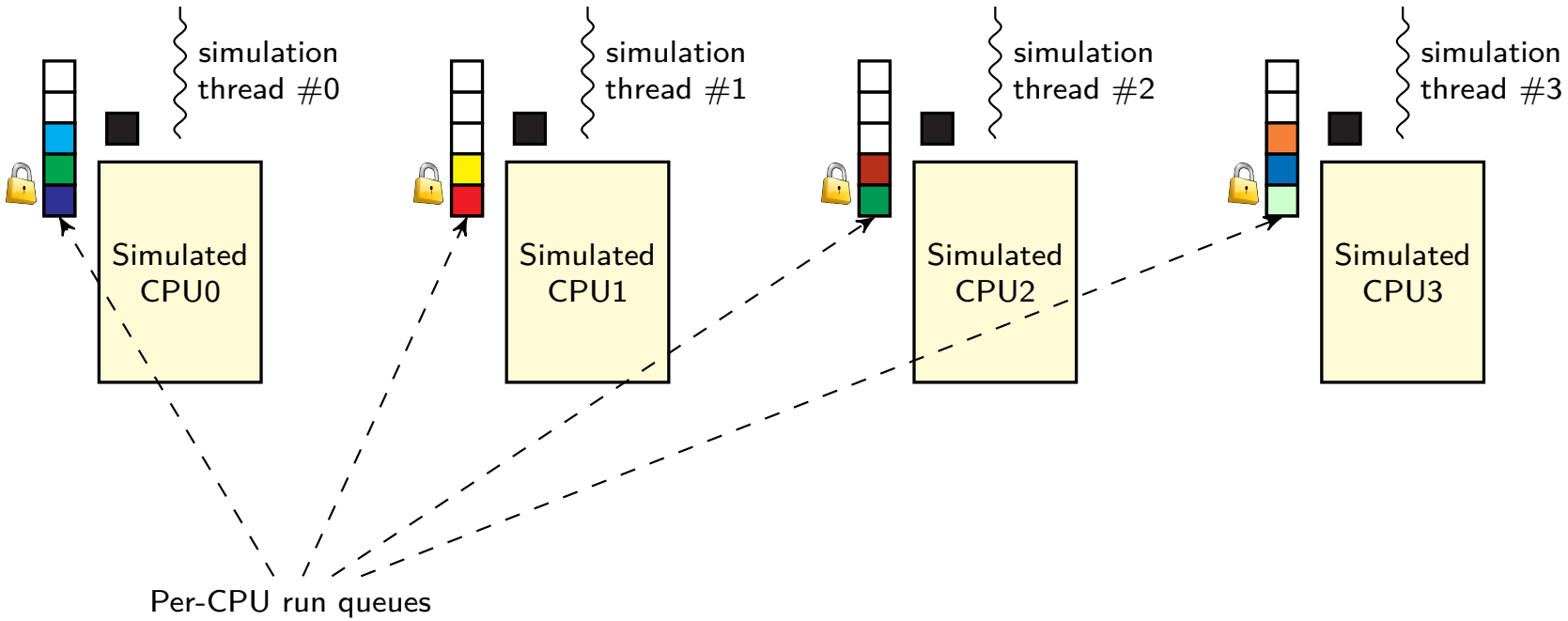
Diseño del simulador

Consta de 2 componentes

1. **Planificador genérico** (`sched.c`)
   - Realiza acciones genéricas por cada ciclo de simulación
     - Equilibra la carga entre CPUs
     - Actualiza estados de las tareas y sus tiempos de ejecución, espera ...
2. **Clases de planificación**
   - 2 clases (RR y SJF) en la versión inicial del simulador
     - Ficheros `sched_rr.c` y `sched_sjf.c`
   - Cada clase implementa un algoritmo de planificación específico
     1. Selecciona la siguiente tarea a ejecutar
     2. Decide cuándo *expropiar* a una tarea
     3. Gestiona la cola de tareas de cada CPU
   - El simulador permite añadir nuevas clases (algoritmos)
     - Cada clase implementa la interfaz `struct sched_class`
     - La clase de planificación *activa* se selecciona al arrancar el simulador (Ejemplo: `./schedsim -s SJF -i examples/example1.txt`)

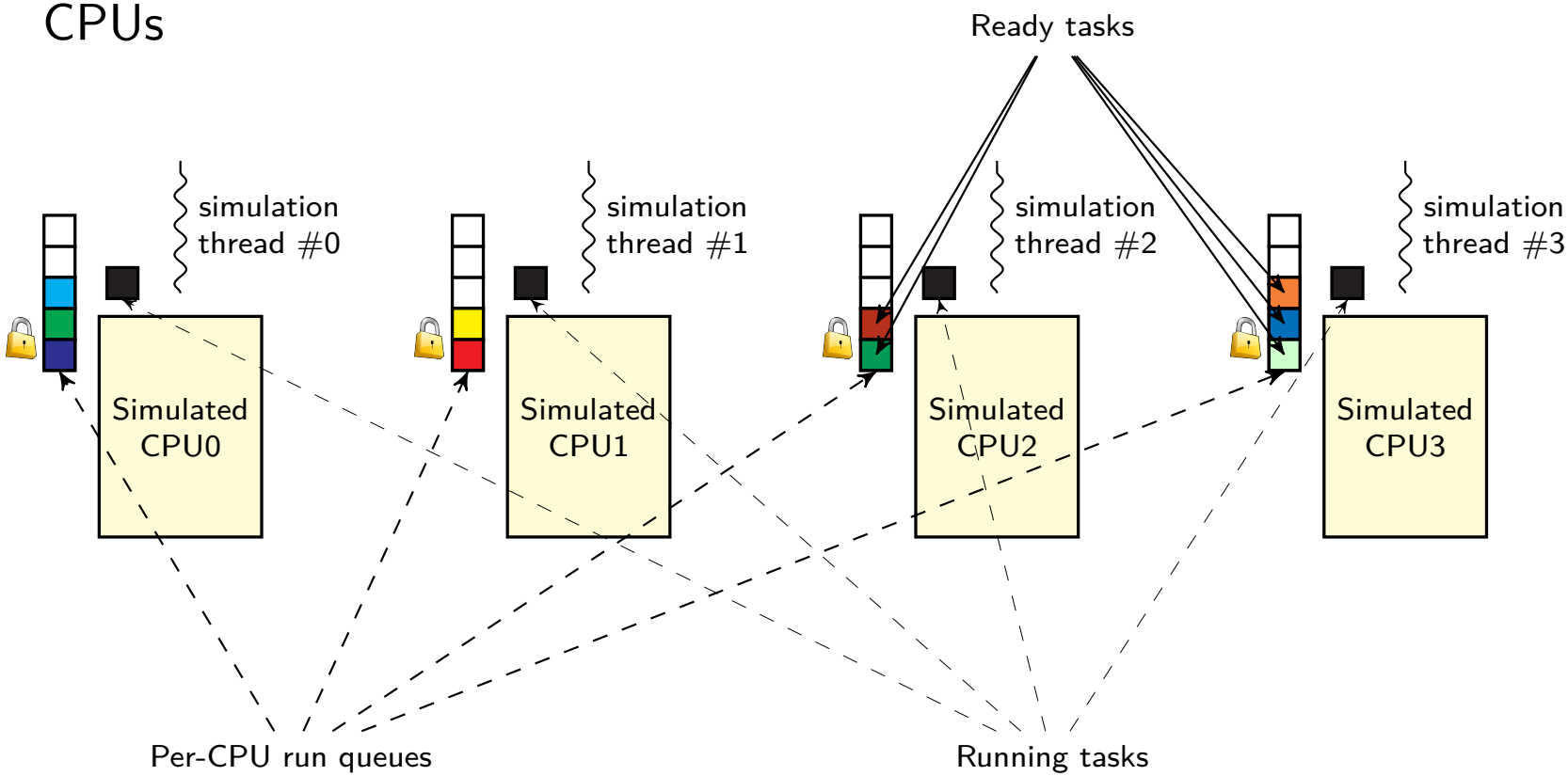
Hilos de simulación vs. tareas

- Existe un hilo real por cada CPU simulada
- Cada CPU (hilo) tiene su cola de tareas listas para ejecutar (*run queue*)
- Cada *run queue* tiene un cerrojo para serializar accesos desde múltiples CPUs

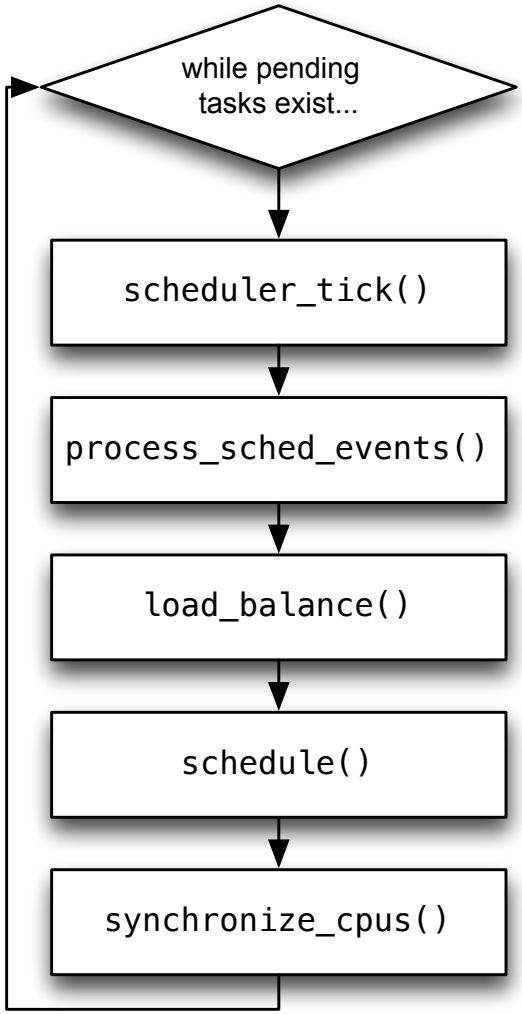


Hilos de simulación vs. tareas

- Existe un hilo real por cada CPU simulada
- Cada CPU (hilo) tiene su cola de tareas listas para ejecutar (*run queue*)
- Cada *run queue* tiene un cerrojo para serializar accesos desde múltiples CPUs

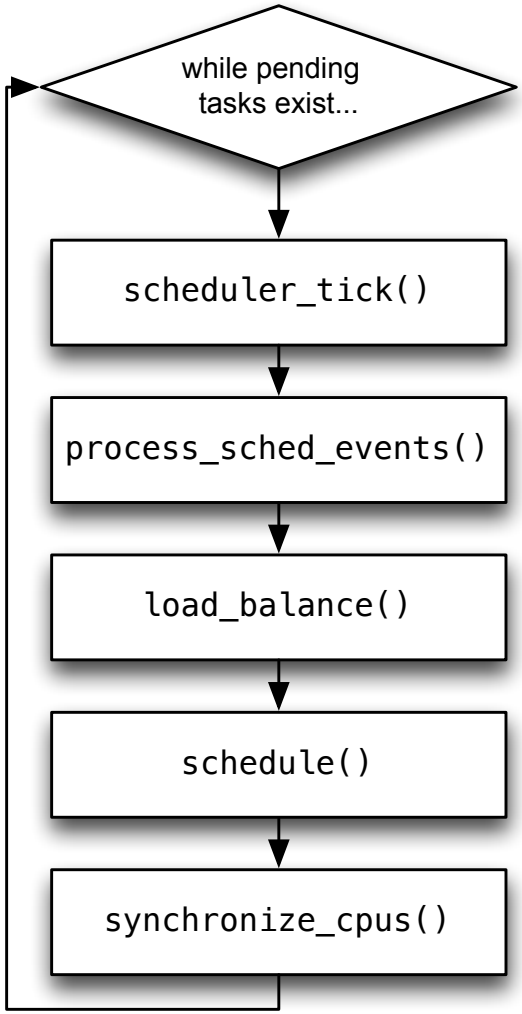


Ciclo de simulación (I)



- Cada hilo de simulación ejecuta este bucle mientras le queden tareas pendientes
  - `sched_cpu()` en `sched.c`
- Una iteración del bucle es equivalente a un *tick* del planificador

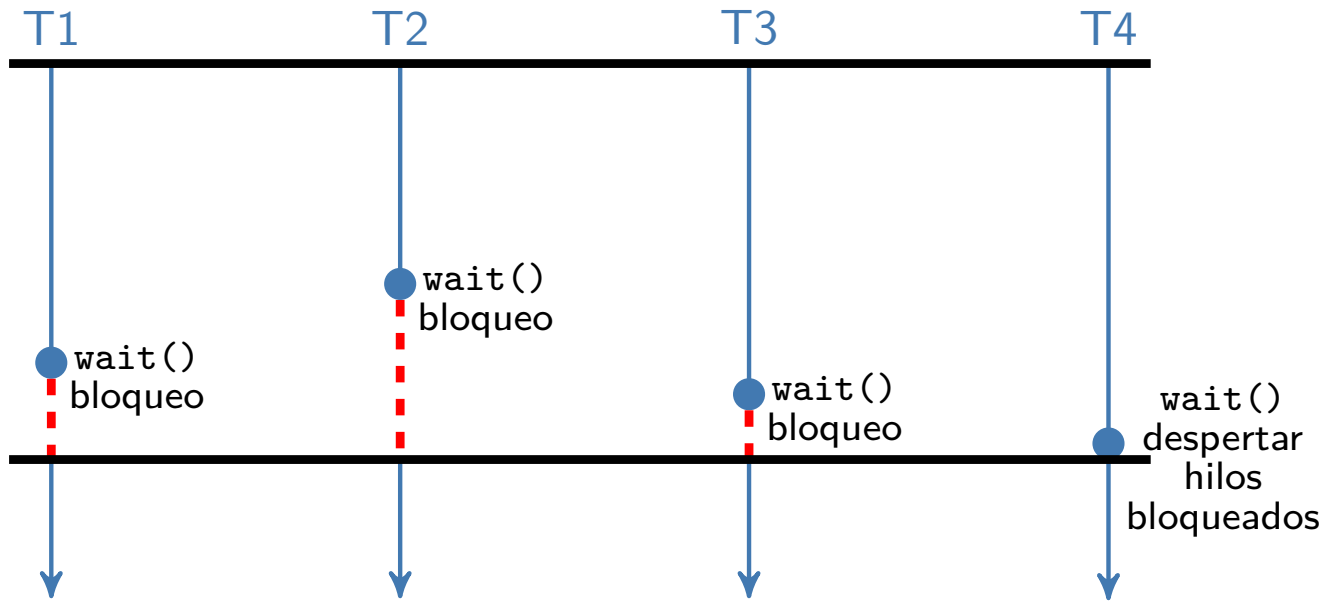
Ciclo de simulación (II)



1. Procesamiento de *tick*
   - Invocar operación `task_tick()` de la CP
   - La clase puede solicitar la expropiación de la tarea en ejecución
2. Despertar tareas bloqueadas/nuevas que estarán listas para ejecutar en el próximo ciclo
3. Equilibrar la carga si es necesario
4. Si CPU estaba idle o tarea en ejecución marcada para expropiar:
   - Seleccionar una nueva tarea para ejecutar (operación `pick_next_task()` de la CP)
   - Si se seleccionó nueva tarea → cambio de contexto
5. Esperar a que las demás CPUs finalicen su ciclo de simulación
   - Se usa una barrera de sincronización

Barrera de sincronización

- Mecanismo de sincronización con una operacion atómica: `wait()`
  - Al crear una barrera es preciso indicar cuántos hilos se sincronizarán en la misma
- Todos los hilos invocan `wait()` para sincronizarse en un mismo punto del código



Barrera de sincronización: implementación

- Por defecto, el simulador usa la implementación de barreras proporcionada por POSIX threads (`pthread_barrier_t`)
- Como parte obligatoria de la práctica será necesario crear una implementación alternativa para la barrera de sincronización

Posible implementación

- 2 contadores
  - `max_threads`: # de hilos que usan la barrera
  - `nr_threads_arrived` : # de hilos que han llegado a la barrera
- 1 mutex (para serializar acceso a los contadores)
- 1 variable condición (para bloquear a los hilos en la barrera)

Barrera de sincronización: implementación

```
typedef struct {
    pthread_mutex_t mutex; /* Barrier lock */
    pthread_cond_t cond; /* Condition variable where threads remain blocked */
    int nr_threads_arrived; /* Number of threads that reached the barrier */
    int max_threads;    /* Number of threads that synchronize at the barrier.
                           (This value is set up upon barrier creation, and must
                                not be modified afterwards) */
}custom_barrier_t;
```

Interbloqueo potencial si ocurre lo siguiente...

1. `max_threads - 1` se bloquean en la barrera (`wait()`)
2. El último thread *UT* llega a la barrera:
   - prepara la barrera para la siguiente invocación de `wait()`: `nr_threads_arrived=0;`
   - despierta a los hilos bloqueados
3. *UT* retorna de `wait()` e invoca `wait()` de nuevo
   - De este modo, `nr_threads_arrived` se actualiza antes de que todos los hilos tengan oportunidad de retornar de `wait()`

Barrera de sincronización: implementación

- Para solucionar este problema, la implementación de la barrera distinguirá entre barreras "pares" e "impares"
  - Para ello, se añade un campo `cur_barrier` (0 o 1) y se mantienen contadores de hilos privados para cada caso (barrera par o impar)
  - El último hilo que alcanza la barrera debe actualizar `cur_barrier`

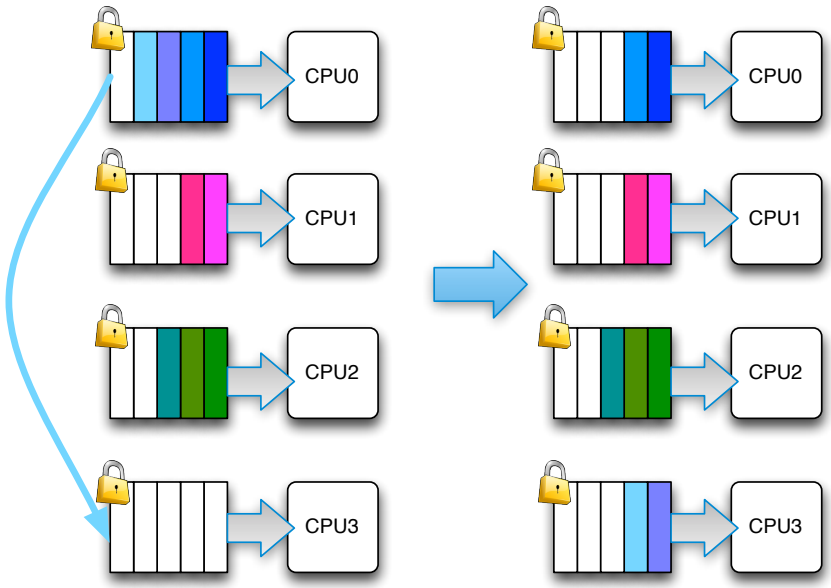
Implementación robusta

```
typedef struct {
   pthread_mutex_t mutex; /* Barrier lock */
   pthread_cond_t cond; /* Condition variable where threads remain blocked */
   int nr_threads_arrived[2]; /* Number of threads that reached the barrier.
                              [0] Counter for even barriers, [1] Counter for odd
                                   barriers */
   int max_threads;     /* Number of threads that synchronize with the barrier.
                              (This value is set up upon barrier creation, and must
                                   not be modified afterwards) */
   unsigned char cur_barrier; /* Field to indicate whether the current barrier is
        an even (0) or an odd (1) barrier */
}sys_barrier_t;
```

Equilibrado de carga

- Se ejecuta de forma distribuida desde cada CPU
  - Se realiza periódicamente o si CPU está *idle*
  - Función `load_balance()`

- El hilo de la CPU más cargada trata de llevar trabajo a la CPU más descargada
- El hilo de la CPU menos cargada trata de *robar* tareas a la CPU más cargada



- Puede ocurrir en paralelo: posible *deadlock*
  - Implementación específica, similar al *Problema de los filósofos*
  - Se adquiere primero el cerrojo de la CPU de mayor número y luego el de la de menor número

Descriptor de la tarea

task_t

```
typedef struct{
    int task_id;
    char task_name[MAX_TASK_NAME];
    exec_profile_t task_profile; /* Task behavior */
    int prio;                    /* Priority */
    task_state_t state;          /* Task state */
    ...
    int runnable_ticks_left;     /* Number of ticks the task
                                    has to complete till blocking or exiting */
    ...
    bool on_rq;             /* flag to indicate if the task is on the rq or not */
    unsigned long flags;       /* generic flags field */
    void* tcs_data;          /* Pointer enabling a scheduling class
                                to store private data if needed */
    ...
}task_t;
```

Flags asociados a una tarea (sched.h)

```
#define TF_IDLE_TASK 0x1 /* Active for the idle task */
/* Enable to indicate that the task must be inserted at the beginning
   of the task list rather than at the end */
#define TF_INSERT_FRONT 0x2
```

run queue (una por CPU)

runqueue_t

```
typedef struct{
    slist_t tasks;     /* runnable task queue (doubly-linked list) */
    task_t* cur_task;   /* Pointer to the currently running task */
    task_t idle_task; /* This CPU's idle task */
    bool need_resched;  /* This flag must be set to TRUE when the
                                sched class wants to preempt the current
                                    task */
    int nr_runnable;  /* Number of runnable task in this CPU
                                -> Note that current is not on the RQ */
    int next_load_balancing;
    void* rq_cs_data;    /* Pointer enabling a scheduling class
                                        to store private data if needed */
    pthread_mutex_t lock; /* Runqueue lock */
}runqueue_t;
```

Listas doblemente enlazadas

Implementación de cola de tareas (slist_t)

```
/* Operaciones básicas */
void* head_slist (slist_t* slist); /* Devuelve el primer elemento de la lista */
void* tail_slist (slist_t* slist); /* Devuelve el último elemento de la lista */
int is_empty_slist (slist_t* slist); /* Devuelve !=0 si lista está vacía */
int size_slist (slist_t* slist);  /* Devuelve número de elementos de la lista */
void remove_slist (slist_t* slist, void* item); /* Eliminar elemento de la lista */
void insert_slist (slist_t* slist, void* item); /* Inserción al final de la
     lista */

/* Operaciones de inserción ordenada
 * (Reciben como parámetro una función de comparación.
 * Consultar ejemplo de uso en sched_sjf.c)
 */
void sorted_insert_slist(slist_t* slist, void* object, int ascending,
      int (*compare)(void*,void*));
void sorted_insert_slist_front(slist_t* slist, void* object, int ascending,
   int (*compare)(void*,void*));
```

Interfaz de la clase de planificación

sched_class

```
typedef struct sched_class {
    /* Operaciones de inicialización/destrucción de la clase */
    void (*sched_init)(void);
    void (*sched_destroy)(void);

    /* Se invoca al crear una nueva tarea */
    void (*task_new)(task_t* t);
    /* Se invoca cuando una tarea termina su ejecución */
    void (*task_free)(task_t* t);

    /* Devuelve y desencola la siguiente tarea a ejecutar en CPU especificada.
       Si no hay ninguna tarea en la cola, devuelve NULL. */
    task_t* (*pick_next_task)(runqueue_t* rq,int cpu);

    /* Se invoca para encolar una tarea
      - Si tarea se acaba de despertar, migrar o es nueva (runnable==0)
    */
    void (*enqueue_task)(task_t* t,int cpu, int runnable);

    /* Procesamiento de tick de la tarea en ejecución (T)
      - Si es preciso, desencadena expropiación de T (rq->need_resched=TRUE;)
       * Al hacer esto el planificador genérico invocará operación pick_next_task()
    */
    void (*task_tick)(runqueue_t* rq,int cpu);

    /* Devuelve y desencola una tarea de esta CPU (para ser migrada a otra CPU) */
    task_t* (*steal_task)(runqueue_t* rq,int cpu);
}sched_class_t;
```

Implementando un planificador

Pasos para añadir un nuevo planificador

1. Implementar nuevo planificador en un fichero .c nuevo
   - Nombre del fichero: `sched_<nombrePlanificador>.c`
   - Exige implementar la interfaz del planificador (`sched_class`)
2. Modificar Makefile para que se compile el nuevo fichero .c
3. Dar de alta el nuevo planificador en `sched.h`

Ejemplo: Añadir planificador FCFS (1/4)

Añadir nuevo fichero `sched_fcfs.c`

```
#include <sched.h>

static task_t* pick_next_task_fcfs(runqueue_t* rq,int cpu) { … }

static void enqueue_task_fcfs(task_t* t,int cpu, int runnable) { … }

static task_t* steal_task_fcfs(runqueue_t* rq,int cpu) { … }


/* Instantiante the interface:
   operation=associated_function
*/
sched_class_t fcfs_sched={
    .pick_next_task=pick_next_task_fcfs,
    .enqueue_task=enqueue_task_fcfs,
    .steal_task=steal_task_fcfs,
};
```

Ejemplo: Añadir planificador FCFS (2/4)

Makefile para que compile `sched_fcfs.c`

```
TARGET=schedsim
SOURCES=main.c sched.c slist.c barrier.c \
        sched_rr.c sched_sjf.c sched_fcfs.c

OBJECTS=$(patsubst %.c,%.o,$(SOURCES))
MY_INCLUDES=.
HEADERS=$(wildcard $(MY_INCLUDES)/*.h)
OS=$(shell uname)
LDFLAGS=-lpthread
#CFLAGS=-g -Wall
CFLAGS=-g -Wall -DPOSIX_BARRIER

…
```

Ejemplo: Añadir planificador FCFS (3/4)

Registrar nuevo planificador en `sched.h`

```
/* Scheduling class descriptors */
extern sched_class_t rr_sched;
extern sched_class_t sjf_sched;
extern sched_class_t fcfs_sched;

/* Numerical IDs for the available scheduling algorithms */
enum {
    RR_SCHED,
    SJF_SCHED,
    FCFS_SCHED,
    NR_AVAILABLE_SCHEDULERS
};

typedef struct sched_choice {
    int sched_id;
    char* sched_name;
    sched_class_t* sched_class;
}sched_choice_t;

/* This array contains an entry for each available scheduler */
static const sched_choice_t available_schedulers[NR_AVAILABLE_SCHEDULERS]={
    {RR_SCHED,"RR",&rr_sched},
    {SJF_SCHED,"SJF",&sjf_sched},
    {FCFS_SCHED,"FCFS",&fcfs_sched}
};
```

Ejemplo: Añadir planificador FCFS(4/4)

- Para comprobar que el planificador se ha añadido correctamente, consultar listado de planificadores disponibles (opción -L)

Terminal

```
debian:P3 osuser$ make clean
...
debian:P3 osuser$ make
gcc  -g -Wall -DPOSIX_BARRIER  -I. -c main.c -o main.o  -Wall
gcc  -g -Wall -DPOSIX_BARRIER  -I. -c sched.c -o sched.o  -Wall
gcc  -g -Wall -DPOSIX_BARRIER  -I. -c slist.c -o slist.o  -Wall
gcc  -g -Wall -DPOSIX_BARRIER  -I. -c barrier.c -o barrier.o  -Wall
gcc  -g -Wall -DPOSIX_BARRIER  -I. -c sched_rr.c -o sched_rr.o  -Wall
gcc  -g -Wall -DPOSIX_BARRIER  -I. -c sched_sjf.c -o sched_sjf.o  -Wall
gcc  -g -Wall -DPOSIX_BARRIER  -I. -c sched_fcfs.c -o sched_fcfs.o  -Wall
gcc  -o  schedsim main.o sched.o slist.o barrier.o sched_rr.o sched_sjf.o
 sched_fcfs.o -lpthread
debian:P3 osuser$ ./schedsim -L
Available schedulers:
RR
SJF
FCFS
debian:P3 osuser$
```

Contenido

Parte obligatoria

Cambios en el código del simulador

1. Crear planificador **FCFS** (no expropiativo)
   - Implementación en nuevo fichero `sched_fcfs.c`
   - Código muy parecido al del RR (FCFS + *timeslices*)
2. Crear planificador **expropiativo** basado en prioridades
   - Implementación en nuevo fichero `sched_prio.c`
   - Basarse en el código del algoritmo SJF expropiativo (`sched_sjf.c`)
   - No olvidar invocar el simulador con opción -p (modo expropiativo)
3. Implementar una barrera de sincronización usando cerrojos y variables condicionales
   - Completar el fichero `barrier.c` (funciones `sys_barrier_init()`, `sys_barrier_destroy()` y `sys_barrier_wait()` de la rama `#else`)
   - Modificar el *Makefile* para evitar que se declare la macro `POSIX_BARRIER`

Parte obligatoria (script)

Script BASH `test.sh`

- Se simulará un ejemplo dado para todos los planificadores implementados y todos los números de CPUs posibles (hasta el máximo indicado)
  - La especificación completa del script se encuentra en el guión de la práctica

Exige usar dos nuevas características del shell

1. Bucles `for` (consultar sintaxis en transparencias sobre BASH)
2. Comando interno `read` para leer una línea de la entrada estándar y guardarla en una variable

Terminal

```
debian:P3 osuser$ read variable
line of text typed with the keyboard
debian:P3 osuser$ echo $variable
line of text typed with the keyboard
debian:P3 osuser$
```

Parte opcional (I)

- Implementar barrera de sincronización usando 2 semáforos POSIX, en lugar de un mutex y una variable condición

```
typedef struct {
  sem_t mtx;  /* It should be initialized with c=1 (to enforce mutual exclusion) */
  sem_t queue; /* It should be initialized with c=0 (Used as a wait queue) */
  int nr_threads_arrived[2]; /* Number of threads that reached the barrier.
                  [0] Counter for even barriers, [1] Counter for odd barriers */
  int max_threads;   /* Number of threads that synchronize with the barrier.
                        (This value is set up upon barrier creation, and must not
                             be modified afterwards) */
  unsigned char cur_barrier; /* Field to indicate whether the current barrier is an
        even (0) or an odd (1) barrier */
}sys_barrier_t;
```

Parte opcional (II)

- Esta implementación alternativa debe crearse en `barrier.c` (definición de funciones) y `barrier.h` (declaración de tipo de datos)
- Elegir barrera con semáforos al compilar sólo si está definido el símbolo de preprocesador `SEM_BARRIER`
  - El símbolo se activará si compilamos con la opción `-DSEM_BARRIER` (modificar Makefile)

```
#ifdef SEM_BARRIER
    <código de la implementación con semáforos>
#else
    <código de la implementación con mutex y variable condición>
#endif
```

Entrega de la práctica

- Hasta el 9 de enero a las 15:55h
- Para realizar la entrega de cada práctica de la asignatura debe subirse un único fichero ".zip" o ".tar.gz" al Campus Virtual
  - Ha de contener todos los ficheros necesarios para compilar la práctica (fuentes + Makefile).
  - Debe ejecutarse "make clean" antes de generar el fichero comprimido
  - Nombre del fichero comprimido:

    `L<num_laboratorio>_P<num_puesto>_Pr<num_prác>.tar.gz`

Estructura entrega (en fichero .zip o .tar.gz)