Diseño y Desarrollo de Sistemas de Tiempo Real

Alfons Crespo

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Clases básicas

- Task
- Tasks
- Partition
- Partitions
- Core
- Cores
- Sistema
- Heap
- BinPacking

Clases básicas

- Diseño
  - Clase
  - Diccionario de clases

- Ejemplo:
  - Tarea: define un objeto de la clase Tarea
  - Tareas: define una diccionario de tareas a las que se puede accede por identificador

Clase: Tarea

- Atributos:
  - Identificador (string)
  - Periodo (entero)
  - Plazo (entero)
  - Offset (entero)
  - Computo (entero)
  - Particion (entero)
  - Utilización (float)

tid: identificador, nombre de la Tarea
period: periodo de la tarea
plazo: plazo de la tarea
wcet: tiempo de peor caso
util: utilizacion
partId: Identificador de la particion

- Operaciones
  - Crear tarea(nombre,periodo, plazo, computo, offset) => identificador
  - PonerPeridodo(id, periodo)
  - Asignar a Partitcion
  - Asignar a Core
  - Utilizacion
  - Print

Task

```
class Task:

# Task: se modela mediante una serie de atributos
  #    tid: identificador, nombre de la Tarea
  #    period: periodo de la tarea
  #.   deadLine: plazo
  #    wcet: tiempo de peor caso
  #.   util: utilizacion
  #    partId: Identificador de la particion
  # ----------------------------------------------------------------------

#Constructora
    def __init__(self, tid, partId):  # attributos: Id, Period, Deadline, WCET, Util
        self.tid = tid
        self.partId = partId
        self.util = None
        self.period = None
        self.deadLine = None
        self.wcet = None
        self.util = None

#Modificadora
    def taskParams(self, Per, Dead, Wcet, util):  # deadline = period
        self.period = Per
        self.deadLine = Dead
        self.wcet = Wcet
        self.util = util

    def taskUtil(self, util):
        self.util = util

#Operaciones Observadora
    def taskGetParams(self):
        return (self.period, self.deadLine, self.wcet, self.util)

    def taskId(self):
        return self.tid

    def taskPeriod(self):
        return self.period

    def taskDeadline(self):
        return self.deadLine

    def taskWCET(self):
        return self.wcet

    def taskUtil(self):
        return self.util

    def taskPartition(self):
        return self.partId

    def show(self):
        return "Task( " +str(self.tid)+ ", " +str(self.period)+ ", "
+str(self.deadLine)+ ", " +str(self.wcet)+ ", " +str(self.util)+ ", "
+str(self.partId)+" )"
```

Tasks

tid: identificador, nombre de la Tarea
period: periodo de la tarea
deadLine: plazo de la tarea
wcet: tiempo de peor caso
util: utilizacion
partId: Identificador de la particion

Módulo **Tasks**

Mantiene:

- **un diccionario de Tareas con una clave tid y el objeto Task**
- **una lista de identificadores de tareas**
- **Número de tareas**

NoTasks = 0;
tasksDict = {}
taskIds = []

Es una interfaz para acceder a Task

Módulo **Tasks**

Atributos:

```
NoTasks = 0;
tasksDict = {}
taskIds = []
```

tid: identificador, nombre de la Tarea
period: periodo de la tarea
deadLine: plazo de la tarea
wcet: tiempo de peor caso
util: utilizacion
partId: Identificador de la particion

Operaciones:

```
def defineTask(tid, util, partId):  # attributos: Id,
Period, Deadline, WCET, Util
def defineTask(tid, util, per, dead, wcet, partId):  #
attributos: Id, Period, Deadline,
def taskParams(tid, per, dead, wcet, util):
def taskGetParams(tid):
def allTasks():
def taskPeriod(tid):
def taskDeadline(tid):
def taskWCET(tid):
def taskPartition(tid):
def taskUtil(tid):
def taskNumber():
def taskShow(tid):
def taskShowAll():
```

Tasks

```
#module  Tasks:

from Task import Task

NoTasks = 0;
taskDict = {}
taskIds = []

def defineTask(tid, util, partId):  # attributos: Id, Period, Deadline,
WCET, Util
   global NoTasks
   if (not taskDict.has_key(tid)):
      tsk = Task(tid, partId)
      taskDict[tid] = tsk
      taskIds.append(tid)
      NoTasks += 1
      return True
   else:
      return False
```

```
def taskParams(tid, per, dead, wcet, util):
   taskDict[tid].taskParams(per, dead, wcet, util)

def taskGetParams(tid):
   return taskDict[tid].taskGetParams()

def allTasks():
   return taskDict.keys()

def taskPeriod(tid):
   return taskDict[tid].taskPeriod()

def taskDeadline(tid):
   return taskDict[tid].taskDeadline()

def taskWCET(tid):
   return taskDict[tid].taskWCET()

def taskPartition(tid):
   return taskDict[tid].taskPartition()

def taskUtil(tid):
   return taskDict[tid].taskUtil()

def taskNumber():
   len(taskDict)
```

```
def taskShow(tid):
        return taskDict[tid].show()

def taskShowAll():
   st = ""
   for t in (taskIds):
      st += taskDict[t].show()
   return st
```

Validacion de tasks y task

```
#!/usr/bin/python

import sys
import os

import Tasks

#------------------------#
#        MAIN            #
#------------------------#
def main (argv):

   for i in range(10):
      partId = "P0"
      tid = "T"+str(i)
      util = 0.8 + i*0.1
      per = 100 + i
      dead = 100 + 2*i
      wcet = i + 2
      ok = Tasks.defineTask(tid, util, partId)
      if (ok):
         Tasks.taskParams(tid, per, dead, wcet, util)
      else:
         print partId, " Exists"

   alltsk = Tasks.allTasks()
   for tid in (alltsk):
      print Tasks.taskShow(tid)

main (sys.argv)
```

```
python testTasks.py
Task( T8, 108, 116, 10, 1.6, P0 )
Task( T9, 109, 118, 11, 1.7, P0 )
Task( T6, 106, 112, 8, 1.4, P0 )
Task( T7, 107, 114, 9, 1.5, P0 )
Task( T4, 104, 108, 6, 1.2, P0 )
Task( T5, 105, 110, 7, 1.3, P0 )
Task( T2, 102, 104, 4, 1.0, P0 )
Task( T3, 103, 106, 5, 1.1, P0 )
Task( T0, 100, 100, 2, 0.8, P0 )
Task( T1, 101, 102, 3, 0.9, P0 )
```

Heap

- Python suministra la cola con política heap
- Se crea importando heapq
  - from heapq import heappush, heappop
- Usaremos 2 funciones básicas:
  - heappush: añade un element a la cola
  - heappush(cola, (clave, item))
  - Ejemplo:
    - readyQueue = []
    - heappush(readyQueue, ((prio), (tid, period, relDead, absDead, wcet, 0, nActiv + 1)))

Heap

- Usaremos 2 funciones básicas:
  - heappush: añade un element a la cola
  - heappop: extrae el element de la cola que está en la Cabeza
    - elm = heappop(cola)
    - (clave, item) = heappop(cola)
    - ((prio), (cTaskId, period, relDead, absDead, wcet, texec, nActiv)) = heappop(readyQueue)

Clase: Particion

- Atributos:
  - Identificador (string)
  - Numero de tareas (Int)
  - Utilización (float)
  - Nivel de criticidad (int)
  - Array de tareas (Lista de tareas) [tid o tareas objeto)
  - Core (int)
- Operaciones
  - Crear Particion(nombre, Criticidad) => identificador
  - AnyadirTarea(tid)
  - NoTareas => numero de tareas de la particion
  - NivelCriticidad => Criticidad Nivel
  - Asignar a Core(cid)
  - EnqueCore => core asignado
  - Utilizacion => utilizacion
  - ListaTareas => lista de identificadores de tareas o objetos tarea
  - Print

Modela la tarea: atributos

pId: Identificador de Partitcion
Clevel: nivel de criticidad
Util: utilización de la particion
UtilEfective: utilización efectiva de la particion
Core: en que core se ejecuta
NoTasks: Número de tareas
TaskList: Lista de identificadores de tarea

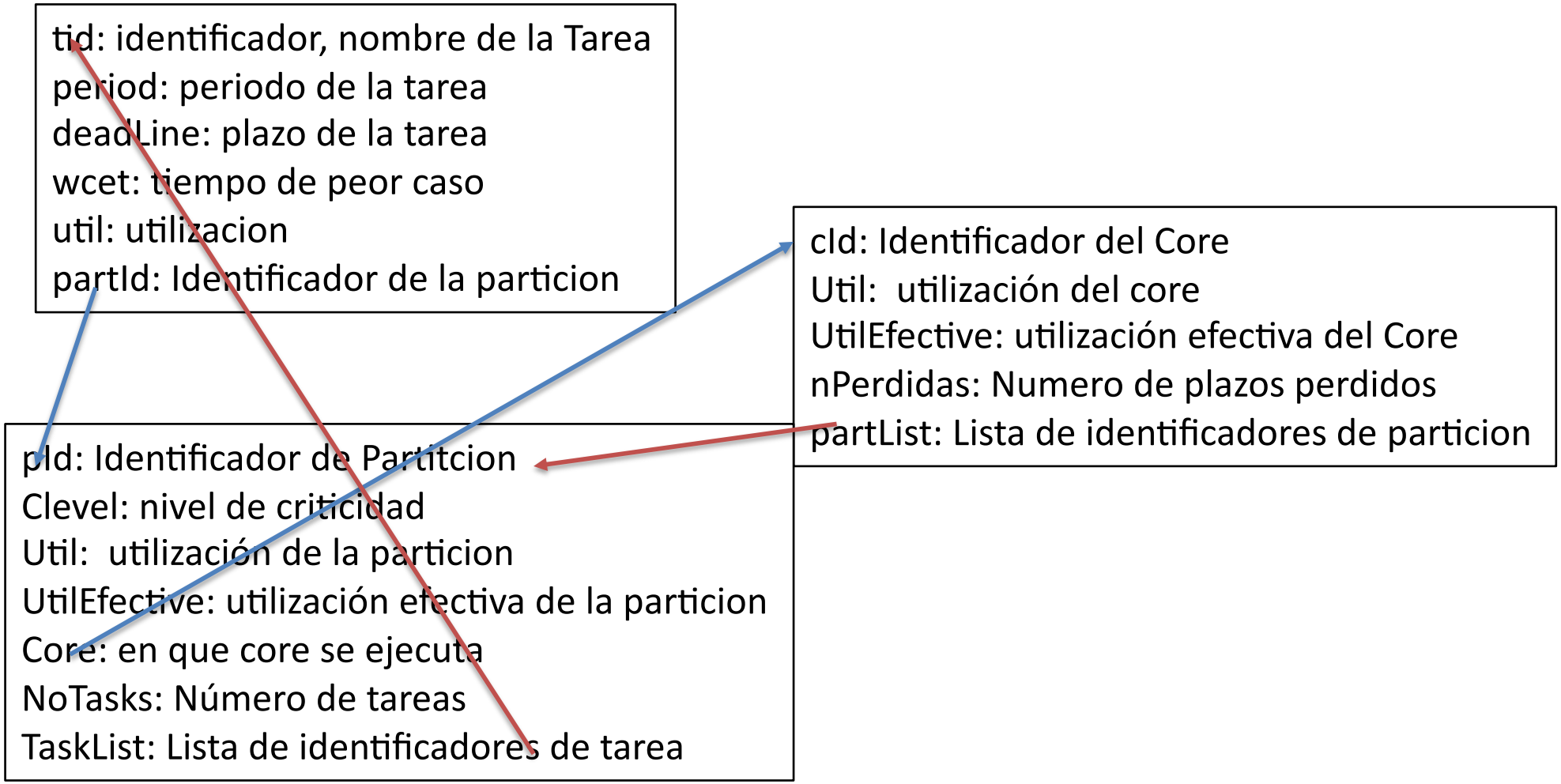
Clase: Core

- Atributos:
  - Identificador (string)
  - Lista de particiones
  - Utilización
- Operaciones
  - Crear Core(nombre) => identificador
  - PonerPeridodo(id, periodo)
  - AnyadirParticion(pid)
  - Utilizacion => utilizacion
  - Print

Modela la core: atributos

cId: Identificador del Core
Util: utilización del core
UtilEfective: utilización efectiva del Core
nPerdidas: Numero de plazos perdidos
partList: Lista de identificadores de particion

Core


tid: identificador, nombre de la Tarea
period: periodo de la tarea
deadLine: plazo de la tarea
wcet: tiempo de peor caso
util: utilizacion
partId: Identificador de la particion
cId: Identificador del Core
Util: utilización del core
UtilEfective: utilización efectiva del Core
nPerdidas: Numero de plazos perdidos
partList: Lista de identificadores de particion
pId: Identificador de Partitcion
Clevel: nivel de criticidad
Util: utilización de la particion
UtilEfective: utilización efectiva de la particion
Core: en que core se ejecuta
NoTasks: Número de tareas
TaskList: Lista de identificadores de tarea

Clases avanzadas

- Analisis de Planificacion
- Asignación de particiones a core
- Ejecución de un sistema (simulador)
- Generador de un plan
- Generador de un sistema

Clase: Sistema

- Depende del lenguaje esta clase identifica los objetos a partir del identificador del core, particion o tarea
- Atributos:
  - Array de cores
  - Array de particiones
  - Array de tareas
- Operaciones
  - CrearSistema
  - AnyadirCore(cid, CoreObjecto)
  - AnyadirPartitcion(pid, ParticionObjeto)
  - AnyadirTareas(tid, TareaObjeto)
  - Print

Clase: BinPacking

- Atributos:
  - NumeroBins
  - Politica (FF, BF, WF) (0,1,2)
  - ListaBins : tantas listas como NumeroBins
  - PesoBins: array de pesos de los Bins
- Operaciones
  - CrearBin(nbins, politica)
  - AnyadirBin(bid, peso) => boolean (true añadido, false no añadido)
  - ListaBin(bid) => lista de ids en el bin bid
  - PesoBin(bid) => peso total en el bin bid
  - Print

Clase: Utils

- Funciones básicas
- Atributos:
  - HiperperidoObjetivo
- Operaciones
  - CrearUtil(HiperperidoObjetivo)
  - Hiperperiodo(array de valores, nvalores) => hiperperiodo
  - AleatorioEntero(min, max) => valor entero aleatorio entre min y max
  - AleatorioReal(min, max) => valor real aleatorio entre min y max
  - AlatorioPeriodoWCET(util) => devuelve el periodo y wcet ajustado a la utilización y con periodo submultiplo de HiperperidoObjetivo
  - UUniFast(nelem, Utotal) => lista de nelem con utilizaciones tal que la suma es Utotal

Modulo Analisis

```
##Module: Analysis


policies = ["RM", "DM", "EDF", "EDFNP"]

def ceil(a, b):
    if ((a % b) == 0):
        return a/b
    else:
        return (a/b) + 1

def utilization(taskSet):
    u = 0.0
    for ts in (taskSet):
        u = u + float(ts[3])/float(ts[1])
    return u

def isRMAplicable(taskSet):

    return True

def WCRT(params, prio):

    Rant = params[prio][3]
    R = Rant
    nt = prio
    while (nt > 0):
        inter = 0
        for i in range(0, nt):
            nexec = ceil(Rant , params[i][1])
            inter += ceil(Rant , params[i][1]) * params[i][3]
        R = params[prio][3] +  inter
        if (R == Rant):
            break
        if (R > params[prio][2]):
            break
        Rant = R
    return R

def exactTestDM(taskParams):
    wrt = []
    for i in range(len(taskParams)):
        w = WCRT(taskParams, i);
        wrt.append([taskParams[i][0], w])
        if (w > taskParams[i][2]):
            return (False, wrt)
    return (True, wrt)
def busyPeriod(params):

    Rant = 0
    for i in range(len(params)):
        Rant = Rant + params[i][3]    # wcet
    next = 0
    while (next <= Rant):
        next = 0
        for i in range(len(params)):
            next += ceil(Rant , params[i][1]) * params[i][3]
        if (next == Rant):
            break
        Rant = next
    return next

def testEDF(taskParams):
    Ut = utilization(taskParams)
    if (Ut <= 1.0):
        return True
    else:
        return False


def schedulabilityTest(policy, ncores, taskParams):
    tparams = []
    print "schedulabilityTest:", policy, ncores, taskParams
    if ((policy == "RM")  and isRMAplicable(taskParams)):
        stparams = sorted(taskParams, key=lambda params: params[1
        return exactTestDM(stparams)
    elif (policy == "DM"):
        stparams = sorted(taskParams, key=lambda params: params[2
        return exactTestDM(stparams)
    elif (policy == "EDF"):
        sched = testEDF(taskParams)
        if (sched):
            primerHueco = busyPeriod(taskParams)
            return sched, primerHueco
        else:
            return sched
    else:
        return "No aplicable"
```

Modulo BinPacking

```
#Module: BinPacking
import random

politicas = ("FF", "NF", "BF", "WF")
politica = ""
criterio = ""
NBins = 0
Bins = {}
Pesos = []
nextBin = 0
init = 0
nFallos = 0

def adjust(v, nd):
   return float(float(round(v * 10**nd)) / 10**nd)

def initBin(metodo, nbins):
   global politica,  NBins, Pesos, init, nextBin, nFallos
   if (metodo in politicas):
      politica = metodo;
   else:
      raise Invalid_Param
   if (nbins >0):
      NBins = nbins
   else:
      raise Invalid_Param
   Pesos = [0.0] * NBins
   for i in range(NBins):
      Bins[i] = []
   init = 1
   nextBin = 0
   nFallos = 0
```

```
def binAdd(item, peso):
# "anade un item a una de las lista con un peso de acuerdo al metodo y criterio
definido en initBins"
   #print "binAdd", item, peso
   if (politica == "FF"):
      ok = binAddFF(item, peso)
   elif (politica == "NF"):
      ok = binAddNF(item, peso)
   elif (politica == "BF"):
      ok = binAddBF(item, peso)
   elif (politica == "WF"):
      ok = binAddWF(item, peso)
   return ok
```

Modulo BinPacking

```
def binAddFF(item, peso):
    #print "addFF", item, peso, NBins
    global Pesos, nFallos
    allocated = 0
    i = 0
    while (allocated == 0) and (i < NBins):
        if ((Pesos[i] + peso) <= 1.0):
            b = Bins[i]
            b.append(item)
            Pesos[i] = adjust(Pesos[i] + peso, 2)
            allocated = 1
        i += 1
    if (allocated == 0):
        nFallos += 1
        return False
    else:
        return True

def binAddNF(item, peso):
    #print "addNF", item, peso
    global nextBin, nFallos
    allocated = 0
    i = nextBin
    n = 0
    while (allocated == 0) and (n < NBins):
        if ((Pesos[i] + peso) <= 1.0):
            b = Bins[i]
            b.append(item)
            Pesos[i] = adjust(Pesos[i] + peso, 2)
            allocated = 1
            nextBin = i
        i = (i + 1) % NBins
        n += 1
    if (allocated == 0):
        nFallos += 1
        return False
    else:
        return True
```

```
def binAddBF(item, peso):
    #print "addBF", item, peso
    global Pesos, nFallos
    allocated = 0
    pcp = []
    for i in range(len(Pesos)):
        tmp = (Pesos[i], i)
        pcp.append(tmp)
    pcp.sort()
    i = 0
    while (allocated == 0) and (i < NBins):
        tmp = pcp.pop(-1)
        p = tmp[0]
        pos = tmp[1]
        if ((p + peso) <= 1.0):
            b = Bins[pos]
            b.append(item)
            Pesos[pos] = adjust(Pesos[pos] + peso, 2)
            allocated = 1
            nextBin = i
        i += 1
    if (allocated == 0):
        nFallos += 1
        return False
    else:
        return True
```

```
def binAddWF(item, peso):
    global Pesos, nFallos
    allocated = 0
    pcp = []
    for i in range(len(Pesos)):
        tmp = (Pesos[i], i)
        pcp.append(tmp)
    pcp.sort()
    i = 0
    while (allocated == 0) and (i < NBins):
        tmp = pcp.pop(0)
        p = tmp[0]
        pos = tmp[1]
        if ((p + peso) <= 1.0):
            b = Bins[pos]
            b.append(item)
            Pesos[pos] = adjust(Pesos[pos] + peso, 2)
            allocated = 1
            nextBin = i
        i += 1
    if (allocated == 0):
        nFallos += 1
        return False
    else:
        return True
```

Modulo. System

```
###Module System
import random
import Utils

import Tasks

def generateSystem(mCores, utilTotal, nTaskMin, nTaskMax):

# define nCores
   nTareas = random.randint(nTaskMin, nTaskMax)
#crea las particiones
   taskParams = []
   if (nTareas > 1):
      utilsTareas = Utils.UUniFast(nTareas, utilTotal)
   else:
      utilsTareas = []
      utilsTareas.append(utilTotal)

   for t in range(nTareas):
      tid =  "T"+str(t)
      ut = utilsTareas[t]
      (per, wcet) = Utils.PeriodWCET(ut)
      #crea las tareas con una particion ficticia
      Tasks.defineTask2(tid, ut, per, per, wcet, "P0")

def allTaskList():
   return Tasks.allTaskParams()

def showSystem():
   for tid in (allTaskList()):
      print tid
```

Diseño Planificador

- Conceptos
- Estructura de datos
- Planificador
- Resultados

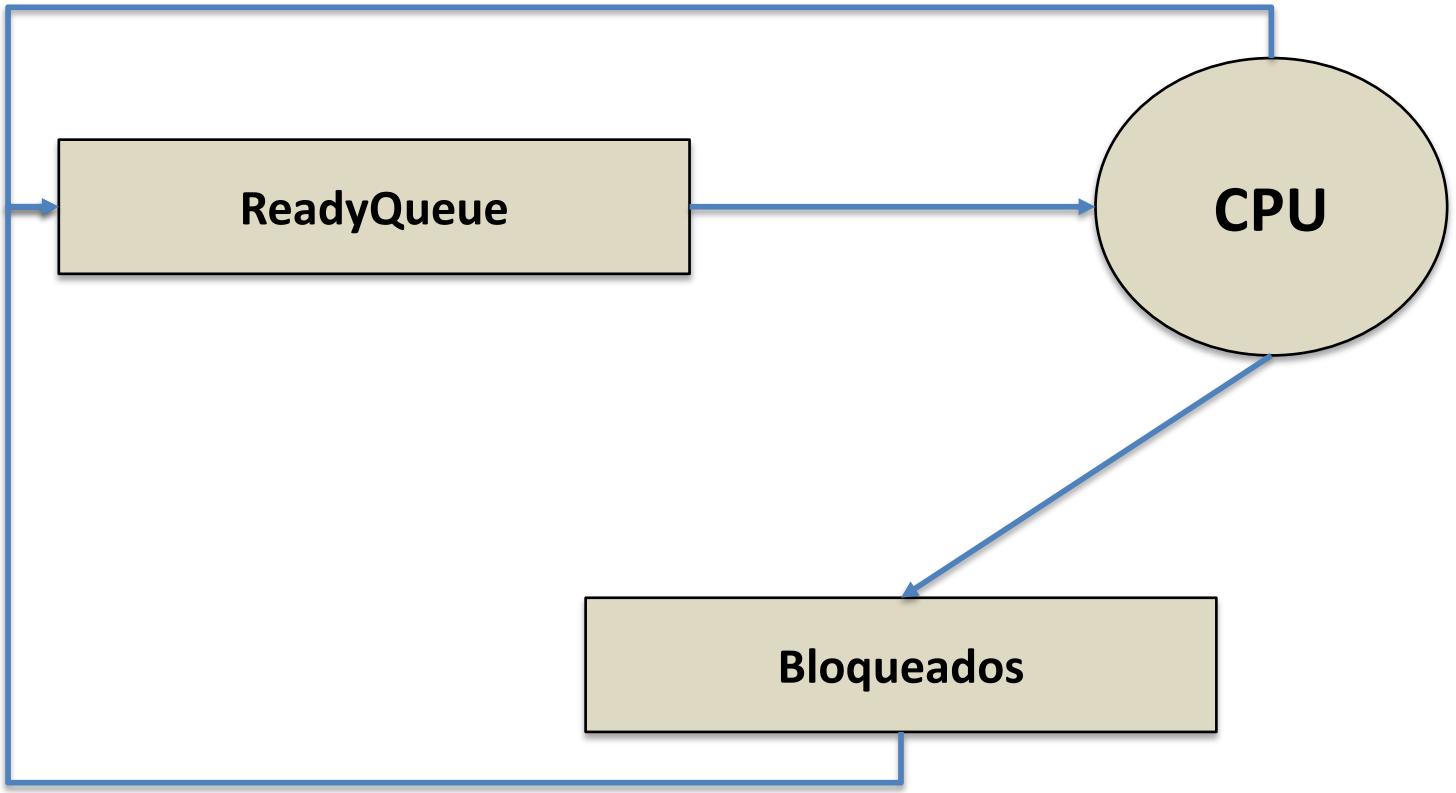
Conceptos

- Se parte de un conjunto de tareas con los parámetros:
  - Id de tarea, Periodo, Plazo, ~~Offset~~, WCET, Prioridad
- Todas las tareas están ordenadas en una estructura de datos ColaBloqueados (heap) por tiempo de activación
  - Offset + k * Periodo (k: número del periodo 0, 1, 2, …)
- Cuando cumplen el tiempo de activación
  - Se sacan de la ColaActivacion
  - Se añaden a la Cola de Preparados
  - Se añaden en la ColaActivación para la siguiente activación

Conceptos

- La ColaPreparados es una lista ordenada por un criterio dependiendo del planificador
  - Prioridad
  - Plazo
  - Laxitud
- Se escoge el primero de la lista
  - Se ejecuta

Diseño Planificador



Estructuras de datos

- ReadyQueue:
  - Cola de tareas preparadas para ser ejecutadas
  - Ordenadas por un criterio:
    - Prioridad
    - Plazo absoluto
  - Estructura Heap
    - La clave puede ser una tupla:
    - El valor puede ser una tupla

Estructuras de datos

- ReadyQueue Ejemplo
  - Añadir a la cola
    - heappush(readyQueue, ((prio), (tid, period, rdead, adead, wcet, 0, nActiv + 1)))
    - La clave es la prioridad
    - El valor: (id de tarea, periodo, plazo relativo, plazo absoluto, wcet, tiempo ejecutado, número de activación)
  - Sacar de la cola:
    - ((prio), (cTaskId, period, rdead, adead, wcet, texec, nActiv)) = heappop(readyQueue)
  - Leer la primera posición sin extraerlo:
    - ((prio), (cTaskId, period, rdead, adead, wcet, texec, nActiv)) = readyQueue[0]
  - Saber el tamaño:
    - len(readyQueue)

Estructuras de datos

- Cola de procesos bloqueados:
  - Cola de tareas esperando para ser activadas cuando cumplan su periodo (periodicas)
  - Ordenadas por un criterio:
    - Tiempo de activación (release time)
  - Estructura Heap
    - La clave puede ser una tupla:
    - El valor puede ser una tupla
  - La denominamos como ColaBloqueados

Estructuras de datos

- ReleaseTime Ejemplo
  - Clave: (tiempo de activación, prioridad/plazo absoluto)
  - Añadir a la cola
    - Al inicio:
      - heappush(ColaBloqueados, ((0, prio), (tid, period, rdead, adead, wcet, 0, nActiv)))
    - Despues:
      - nActiv + 1
      - nxtActiv = nActiv * period
      - adead = nxtActiv + rdead
      - heappush(releaseTime, ((nxtActiv, prio), (cTaskId, period, rdead, adead, wcet, 0, nActiv)))
  - Leer la primera posición sin extraerlo:
    - (time, prio) = ColaBloqueados[0][0]
    - (tid, period, rdead, adead, wcet, texec, nActiv) = ColaBloqueados[0][1]
  - Sacar de la cola
    - elem = heappop(ColaBloqueados)
  - Saber el tamaño:
    - len(ColaBloqueados)

Planificador: inicialización

- Cabecera: def schedRun(ticks):
- Construir una lista con las tareas
  - Global: todas las tareas de todas las particiones
  - Local: una por cada core con las tareas de las particiones asociadas al core
- Prioridades:
  - Ordenar la lista por periodos
    - Si el primer parámetro de la lista es el periodo: tList.sort()
    - Recorrer la lista asignandole prioridades crecientes (1 mayor prioridad, n menor prioridad)
    - Una vez asignada la prioridad, añadirlas a la cola releaseTime con tiempo 0

Planificador

- Inicialización
- Clock = 0
- Mientras clock < nticks
  - Actualizar la ReadyQueue desde la ReleaseTime al tiempo de release
  - Seleccionar la tarea a ejecutar (primera de la ready queue)
  - Incrementar tiempo de ejecución
  - Si termina:
    - Se añade a la releaseTime con tiempo = siguiente activacion
  - Si no termina:
    - Se añade a la readyQueue con valores actualizados
  - Incrementar clock

Planificador: inicialización

- Cabecera: def schedRun(ticks):
- Construir una lista con las tareas
  - Global: todas las tareas de todas las particiones
  - Local: una por cada core con las tareas de las particiones asociadas al core
- EDF:
  - Ordenar la lista por plazos relativos
    - Si el primer parámetro de la lista es el plazo: tList.sort()
    - Añadirlas a la cola releaseTime con tiempo 0

Trazador

```
##Module: Utils
import random

Traces = {}
mcores = 0
Activations = {}
last  =  ""
lastTask = ""

def traceInit(ncores):
    global last
    mcores = ncores
    for i in range(ncores):
            Traces[i]  =  []
    Activations = {}
      last  =  ""

def traceExecBegin(ncore, time, taskId):
    global last, lastTask
    list = Traces[ncore]
      if  (last  ==  "B"):
       list.append((lastTask, "TE", time))
    list.append((taskId, "TB", time))
    #print "TB", time, taskId
    Traces[ncore] = list
    lastTask = taskId
      last  =  "B"
    if (Activations.has_key(taskId)):
       Activations[taskId] = Activations[taskId] + 1
      else:
       Activations[taskId] = 1

def traceExecEnd(ncore, time, taskId):
    global last, lastTask

    list = Traces[ncore]
    list.append((taskId, "TE", time))
    Traces[ncore] = list
    #print "TE", time, taskId
      last  =  "E"


def traceShow(ncore):
      res  =  ""
    trz = Traces[ncore]
    for i in range(len(trz)):
          if ("TB" in  trz[i]):
                startTime  =  trz[i][2]
                res  +=  trz[i][0]  +  "  ["  +  str(startTime)
          elif ("TE" in  trz[i]):
                endTime  =  trz[i][2]
        duration = endTime - startTime
                res  +=  ",  "  +  str(duration)  +  "]\n"
    print res
    print Activations
```

trabajos

1. **Planificador monocore con EDF**
2. **Planificador local multicore Prioridades RM**
    1. **Generar un sistema con 4 cores, utilización = 3.2**
    2. **Definir una política de reparto en cores: (bin packing FF, NF, BF, WF)**
    3. **Ejecutar cada subconjunto de tareas en un core con distinta política de planificación (DM, EDF)**
3. **Planificador global multicore Prioridades fijas RM, EDF**
    1. **Generar un sistema para 4 cores**
    2. **Ejecutar el sistema con los 4 recursos**

En 1:

Que imprima el peor tiempo de respuesta observado en la simulación tanto para RM,DM y EDF

En 2 :

Comparar los resultados de planificación de un mismo sistema con las distintas políticas de asignación a core y planificación

En 3:

Analizar los cambios de contexto que se producen bajo RM y EDF.