Scheduling (4)

Fair Share Implementation (1)

```python
#!/usr/bin/python
processes = [1,2,3]                         # three processes
process_queue = [1,2,3]
base_priorities = {1:60, 2:60, 3:60}        # with equal priorities
groups = [ [1], [2,3] ]                     # two groups
weight = [0.5, 0.5]                         # with related weights
runtime = 30                                # how long to run them?

# Initialization
proccount = len (processes)
groupcount = len (groups)
activity_log = []
seconds = 0
priorities = {}       # empty associative lists
proc_cpu_util = {}
group_cpu_util = []   # not possible with groups...

for p in processes:
  priorities[p] = base_priorities[p]
  proc_cpu_util[p] = 0

for i in range(0,groupcount):
  group_cpu_util.append(0)
```

```python
def print_status():
  print "%4d |" % seconds,
  for p in processes:
    prio = priorities[p]
    pcpu = proc_cpu_util[p]
    gcpu = group_cpu_util
      [group_of(p)]
    print "%3d %3d %3d  | "
      % (prio,pcpu,gcpu),
  return
# end print_status()
```

Fair Share Implementation (2)

```python
def find_min_priority():
  # find process with lowest
  # priority; if there are several
  # with lowest: first of them
  minval = 9999
  for p in process_queue:
    if priorities[p] < minval:
      minval = priorities[p]
      minproc = p
  # move process to the end of
  # the queue
  process_queue.remove(minproc)
  process_queue.append(minproc)
  return minproc
# end find_min_priority()

def group_of(p):
  for i in range (0,groupcount):
    if p in groups[i]: return i
# end group_of()
```

```python
def recalculate():
  # recalculate priorities
  proc_cpu_util[active] += 60
  g = group_of (active)
  group_cpu_util[g] += 60

  for g in range(0,groupcount):
    group_cpu_util[g] /= 2

  for p in processes:
    g = group_of (p)
    proc_cpu_util[p] /= 2     # halve
    prio = base_priorities[p]                \
      + int( proc_cpu_util[p] / 2 )          \
      + int( group_cpu_util[g] /             \
                 (4 * weight[g]) )
    priorities[p] = prio
  return
# end recalculate()
```

Fair Share Implementation (3)

```python
# begin main()
print "Time |",
for p in processes:
  print "Process %2d    | " % p,
print
print_status()

for i in range(0,runtime):
  active = find_min_priority()
  print "-> Scheduler activates p.",
    active
  activity_log.append(active)
  # run active process
  seconds += 1  # count time
  recalculate()
  print_status()

print
```

```python
# display statistics

print

oldprocess = 0
for p in processes:
  st = ""  # empty string
  if group_of(p) != group_of(oldprocess):
    print "NEW GROUP"
  print "Process %1d:" % p,
  for i in range(0,runtime):
    if activity_log[i] == p:
      st += "x"
    else:
      st += "-"
  print st
  oldprocess=p

# end main()
```

Fair Share Implementation (4)

```
> ./fair-share.py
Time | Process   1   |  Process   2  |  Process   3  |
   0 |  60   0    0  |  60    0    0 |  60    0    0 |  -> Scheduler aktiviert P. 1
   1 |  90  30   30  |  60    0    0 |  60    0    0 |  -> Scheduler aktiviert P. 2
   2 |  74  15   15  |  90   30   30 |  75    0   30 |  -> Scheduler aktiviert P. 1
   3 |  96  37   37  |  74   15   15 |  67    0   15 |  -> Scheduler aktiviert P. 3
   4 |  78  18   18  |  81    7   37 |  93   30   37 |  -> Scheduler aktiviert P. 1
   5 |  98  39   39  |  70    3   18 |  76   15   18 |  -> Scheduler aktiviert P. 2
   6 |  78  19   19  |  94   31   39 |  82    7   39 |  -> Scheduler aktiviert P. 1
   7 |  98  39   39  |  76   15   19 |  70    3   19 |  -> Scheduler aktiviert P. 3
   8 |  78  19   19  |  82    7   39 |  94   31   39 |  -> Scheduler aktiviert P. 1
   9 |  98  39   39  |  70    3   19 |  76   15   19 |  -> Scheduler aktiviert P. 2
  10 |  78  19   19  |  94   31   39 |  82    7   39 |  -> Scheduler aktiviert P. 1
  11 |  98  39   39  |  76   15   19 |  70    3   19 |  -> Scheduler aktiviert P. 3
  12 |  78  19   19  |  82    7   39 |  94   31   39 |  -> Scheduler aktiviert P. 1
  13 |  98  39   39  |  70    3   19 |  76   15   19 |  -> Scheduler aktiviert P. 2
  14 |  78  19   19  |  94   31   39 |  82    7   39 |  -> Scheduler aktiviert P. 1
  15 |  98  39   39  |  76   15   19 |  70    3   19 |  -> Scheduler aktiviert P. 3
  16 |  78  19   19  |  82    7   39 |  94   31   39 |  -> Scheduler aktiviert P. 1
  17 |  98  39   39  |  70    3   19 |  76   15   19 |  -> Scheduler aktiviert P. 2
  18 |  78  19   19  |  94   31   39 |  82    7   39 |  -> Scheduler aktiviert P. 1
  19 |  98  39   39  |  76   15   19 |  70    3   19 |  -> Scheduler aktiviert P. 3
  20 |  78  19   19  |  82    7   39 |  94   31   39 |  -> Scheduler aktiviert P. 1
  21 |  98  39   39  |  70    3   19 |  76   15   19 |  -> Scheduler aktiviert P. 2
  22 |  78  19   19  |  94   31   39 |  82    7   39 |  -> Scheduler aktiviert P. 1
  23 |  98  39   39  |  76   15   19 |  70    3   19 |  -> Scheduler aktiviert P. 3
  24 |  78  19   19  |  82    7   39 |  94   31   39 |  -> Scheduler aktiviert P. 1
  25 |  98  39   39  |  70    3   19 |  76   15   19 |

NEW GROUP
Process 1: x-x-x-x-x-x-x-x-x-x-x-x-x
NEW GROUP
Process 2: -x---x---x---x---x---x---
Process 3: ---x---x---x---x---x---x-
```


Fair Share Implementation (5)

More processes in one group

```
processes = [1,2,3,4,5]                 # five processes
base_priorities = {1:60, 2:60, 3:60, 4:60, 5:60}
groups = [ [1], [2,3,4,5] ]             # two groups
weight = [0.5, 0.5]                     # with related weights
```

```
NEW GROUP
Process 1: x-x--x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x
NEW GROUP
Process 2: -x------x-------x-------x-------x-------
Process 3: ---x------x-------x-------x-------x-----
Process 4: ----x-------x-------x-------x-------x---
Process 5: ------x-------x-------x-------x-------x-
```


Fair Share Implementation (6)

Equal-sized groups

```
processes = [1,2,3,4,5,6]                  # six processes
base_priorities = {1:60, 2:60, 3:60, 4:60, 5:60, 6:60}
groups = [ [1,2,3], [4,5,6] ]              # two groups
weight = [0.5, 0.5]                        # with related weights
```

```
NEW GROUP
Process 1: x-----x-----x-----x-----x-----x-----x---
Process 2: --x-----x-----x-----x-----x-----x-----x-
Process 3: ----x-----x-----x-----x-----x-----x-----
NEW GROUP
Process 4: -x-----x-----x-----x-----x-----x-----x--
Process 5: ---x-----x-----x-----x-----x-----x-----x
Process 6: -----x-----x-----x-----x-----x-----x----
```


Fair Share Implementation (7)

Three groups with differently many processes

```
processes = [1,2,3,4,5,6,7]                # seven processes
base_priorities = {1:60, 2:60, 3:60, 4:60, 5:60, 6:60, 7:60}
groups = [ [1], [2,3], [4,5,6,7] ]         # three groups
weight = [0.33, 0.33, 0.33]                # with related weights
```

```
NEW GROUP
Process 1: x--x--x--x--x--x--x--x--x--x--x--x--x--x--x--x--x- (34 Prozent)
NEW GROUP
Process 2: -x-----x-----x-----x-----x-----x-----x-----x-----x (18 Prozent)
Process 3: ----x-----x-----x-----x-----x-----x-----x-----x--- (16 Prozent)
NEW GROUP
Process 4: --x-----------x-----------x-----------x----------- ( 8 Prozent)
Process 5: -----x-----------x-----------x-----------x-------- ( 8 Prozent)
Process 6: --------x-----------x-----------x-----------x----- ( 8 Prozent)
Process 7: -----------x-----------x-----------x-----------x-- ( 8 Prozent)
```

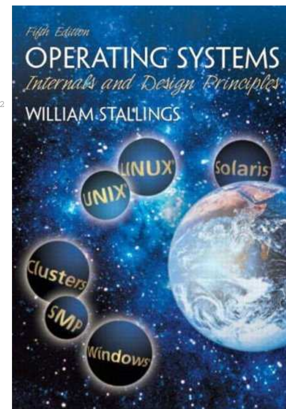
Shares as upper / lower limit

Two possibilities

- **Shares define upper limit:**
  System guarantees that no process receives more CPU time than it deserves (as long as system would not become idle otherwise)
- **Shares define lower limit,**
  i.e. a guaranteed property:
  System assures that each process receives exactly the promised part of the compute time (or more, if possible)

Scheduling on Multi-Processor-Systems

Slides are (partially) based on W. Stallings, ch. 10



Classifications of Multiprocessor Systems

- Loosely coupled multiprocessor
  - Each processor has its own memory and I/O channels
- Functionally specialized processors
  - Such as I/O processor
  - Controlled by a master processor
- Tightly coupled multiprocessing
  - Processors share main memory
  - Controlled by operating system

Independent Parallelism

- Separate applications
- No synchronization
- More than one processor is available
  - Average response time to users is less

Coarse and Very Coarse-Grained Parallelism

- Synchronization among processes at a very gross level
- Good for concurrent processes running on a multiprogrammed uniprocessor
  - Can by supported on a multiprocessor with little change

Medium-Grained Parallelism

- Parallel processing or multitasking within a single application
- Single application is a collection of threads
- Threads usually interact frequently

Fine-Grained Parallelism

- Highly parallel applications
- Specialized and fragmented area

Granularity – overview

granularity	description	synchonisation intervall
fine	parallelism inherent in single instruction streams	< 20
medium	parallel processing within one single application	20-200
coarse	multitasking of concurrent processes in a multiprogramming environment	200-2.000
very coarse	distributed processing across network nodes to form a single computing environment	2.000-1.000.000
independent	several unrelated processes	–

Scheduling

- Assignment of processes to processors
- Use of multiprogramming on individual processors
- Actual dispatching of a process

Assignment of Processes to Processors (1)

- Treat processors as a pooled resource and assign process to processors on demand
- Permanently assign process to a processor
  - Dedicate short-term queue for each processor
  - Less overhead
  - Processor could be idle while another processor has a backlog

Assignment of Processes to Processors (2)

- Global queue
  - Schedule to any available processor
- Master/slave architecture
  - Key kernel functions always run on a particular processor
  - Master is responsible for scheduling
  - Slave sends service request to the master
  - Disadvantages
    - Failure of master brings down whole system
    - Master can become a performance bottleneck

Assignment of Processes to Processors (3)

- Peer architecture
  - Operating system can execute on any processor
  - Each processor does self-scheduling
  - Complicates the operating system
    - Make sure two processors do not choose the same process

Process Scheduling

- Single queue for all processes
- Multiple queues are used for priorities
- All queues feed to the common pool of processors
- Specific scheduling disciplines is less important with more than on processor

Threads

- Executes separate from the rest of the process
- An application can be a set of threads that cooperate and execute concurrently in the same address space
- Threads running on separate processors yields a dramatic gain in performance

Multiprocessor Thread Scheduling (1)

- Load sharing
  - Processes are not assigned to a particular processor
- Gang scheduling
  - A set of related threads is scheduled to run on a set of processors at the same time

Multiprocessor Thread Scheduling (2)

- Dedicated processor assignment
  - Threads are assigned to a specific processor
- Dynamic scheduling
  - Number of threads can be altered during course of execution

Load Sharing

- Load is distributed evenly across the processors
- No centralized scheduler required
- Use global queues

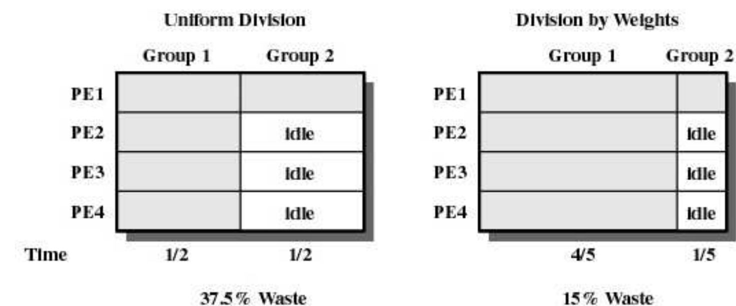
Disadvantages of Load Sharing

- Central queue needs mutual exclusion
  - May be a bottleneck when more than one processor looks for work at the same time
- Preemptive threads are unlikely resume execution on the same processor
  - Cache use is less efficient
- If all threads are in the global queue, all threads of a program will not gain access to the processors at the same time

Gang Scheduling

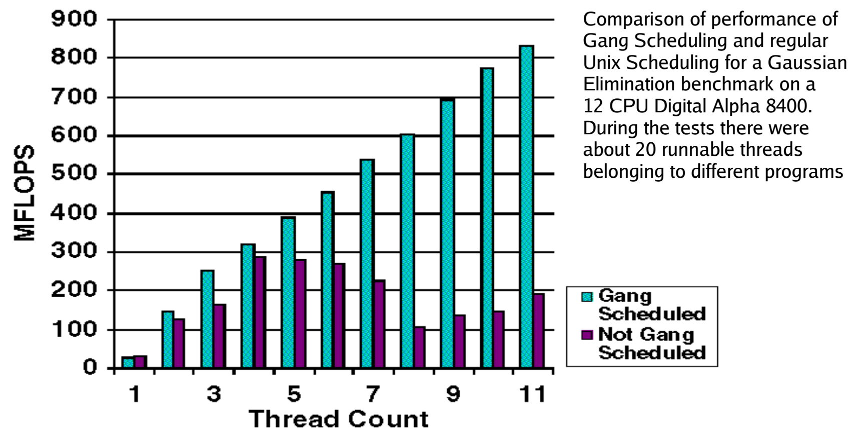
- Simultaneous scheduling of threads that make up a single process
- Useful for applications where performance severely degrades when any part of the application is not running
- Threads often need to synchronize with each other

Scheduling Groups



Scheduling groups with 4 / 1 threads

Gang Scheduling: Performance



Comparison of performance of Gang Scheduling and regular Unix Scheduling for a Gaussian Elimination benchmark on a 12 CPU Digital Alpha 8400. During the tests there were about 20 runnable threads belonging to different programs

Picture: http://www.llnl.gov/asci/pse_trilab/sc98.summary.html

Dedicated Processor Assignment

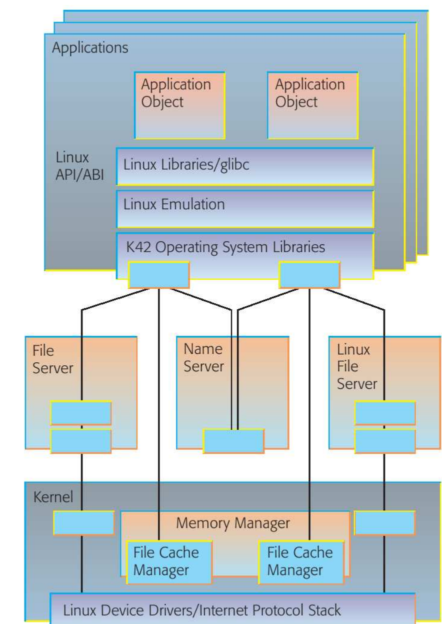
- When application is scheduled, its threads are assigned to a processor
- Some processors may be idle
- No multiprogramming of processors

Dynamic Scheduling

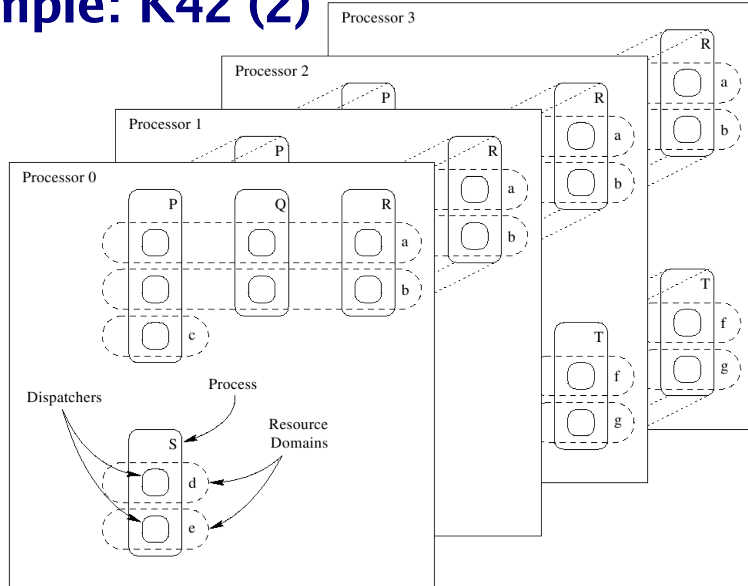
- Number of threads in a process are altered dynamically by the application
- Operating system adjust the load to improve use
  - Assign idle processors
  - New arrivals may be assigned to a processor that is used by a job currently using more than one processor
  - Hold request until processor is available
  - New arrivals will be given a processor before existing running applications

Example: K42 (1)

- Linux-compatible OS for machines with hundreds of CPUs
- object-oriented; OS calls implemented as IPC calls
- two scheduling levels
- thread scheduling: completely in user mode
- kernel scheduler:
  - manages so called dispatchers which in turn manage threads (often: one dispatcher per process);
  - runs independently on each CPU



Example: K42 (2)



Example: K42 (3)

- Thread Migration:
  K42 thread scheduler implements load balancing by migrating threads from busy to idle dispatchers
- Kernel (in special situations) moves dispatchers to different CPUs
- Kernel scheduler activates resource domains (within a domain: dispatchers in a ring)
- also allows Gang Scheduling (see processes P, R)
- Dispatcher can block individual threads (e.g. for page fault or I/O) without blocking itself

Example: K42 (4)

- Process has the choice:
  - real multitasking -> use several dispatchers
  - only programming comfort of threads
    -> one dispatcher is enough
- Several thread libraries for programmers, including POSIX threads
- K42 Scheduler: http://www.research.ibm.com/K42/white-papers/Scheduling.pdf (2002)
- Introduction to K42: http://www.research.ibm.com/journal/sj/442/appavoo.pdf