

- More on Process Management & Context Switching...
  - traditional PCB
  - multi-level (before Solaris 9): PCB, LWP, kernel and user threads (handout)
  - Linux task structure

# Traditional PCB

- Process ID
- user credentials
- runtime info
  - priority, signal mask, registers, kernel stack, resource limitations...
- signal handlers table
- memory map
- file descriptors

# Data structures

data for processor, process, LWP and kernel thread

- per-process structure
  - list of kernel threads associated with process
  - pointer to process address space
  - user credentials
  - list of signal handlers
- per-LWP structure (swappable):
  - user-level registers
  - system call arguments
  - signal handling masks
  - resource usage information
  - profiling pointers
  - pointer to kernel thread and process structure

# Data structures

- kernel thread structure
  - kernel registers
  - scheduling class
  - dispatch queue links
  - pointer to the stack
  - pointers to LWP, process, and CPU structure
- CPU structure:
  - pointer to currently executing thread
  - pointer to idle thread
  - current dispatching and interrupt handling info
  - for the most part architecture independent
- for efficient access to structures, hold pointer to current thread in global register
  - enables access to struct fields with single instruction

- Linux threads and processes represented via task struct
  - collection of pointers to corresponding content
  - threads from same process share FS, MM, signal handlers, open files... (see Silberschatz 4.5.2)
    - kernel code takes care of locking if necessary for shared access
  - all tasks linked into lists, but also accessible via hash on PID
  - if architecture permits (PPC, UltraSparc) current pointer in register

- User level threads
  - PC, SP
  - registers
  - stack
  - thread private/specific data
    - e.g., for `errno`
- Memory footprint/access critical for performance.

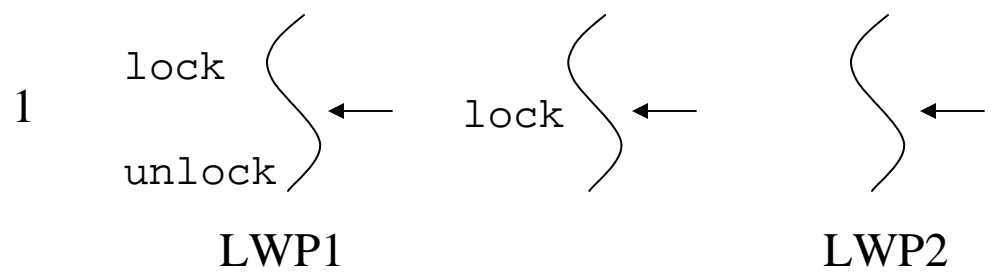
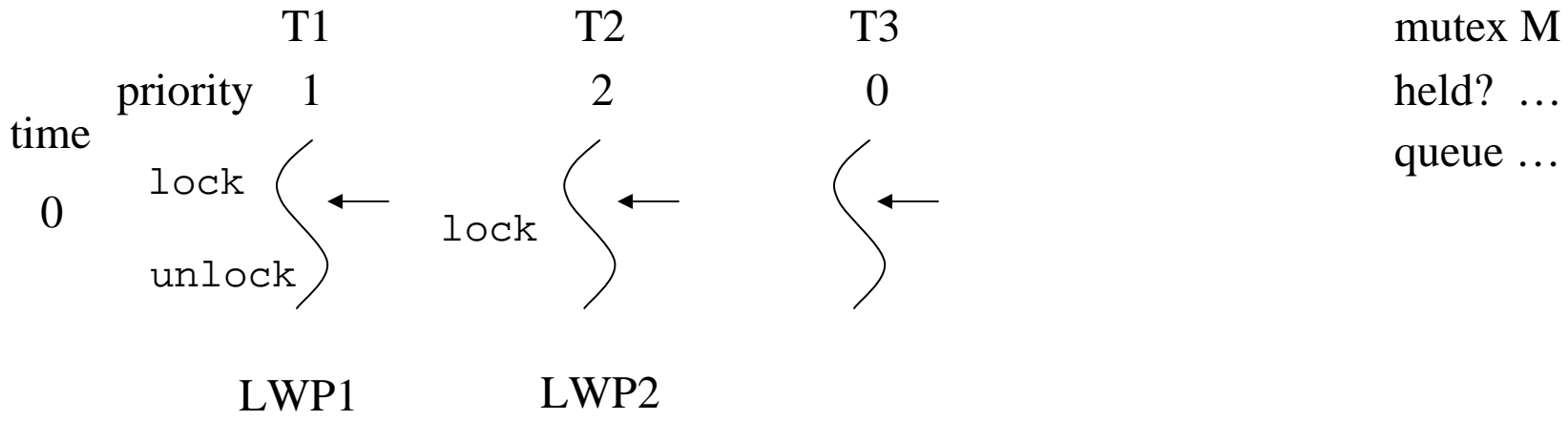
# Process Contention Scope Context Switching

There are four ways to cause a running thread to context switch:

- **Synchronization**
  - the most common way: the thread goes to sleep on a mutex or condition variable
- **Pre-emption**
  - a running thread does something that causes a high-priority thread to become runnable
  - hard to implement entirely in user-space (in SMPs) except in a one-to-one model
- **Yielding**
  - a thread may explicitly yield to another thread of the same priority
- **Time-slicing**
  - threads of the same priority may be context switched periodically

# PCS Contexts Switching

- Time slicing and pre-emption need kernel collaboration (except for pre-emption on uniprocessors)
  - at the very least a signal needs to be sent and/or handled
- *Question:* What happens when the library context switches threads?



2

- Context switch is necessary and nowadays efficient
- Key factor is warm cache
  - architecture support may help
  - “processor affinity” or “cache affinity”

# Scheduling

- Mostly Siberschatz 5.1-5.6
  - (Lewis & Berg Ch. 5 too)
- Scheduling determines who runs next/when
- Dispatcher puts that decision in action
  - switches contexts, switches to user mode, jumps to PC and gives CPU to selected process

- Scheduling decisions occur
  - process switches from running to waiting state
  - process switches from running to ready (something happened, e.g., interrupt, to cause it to stop running)
  - process switches from waiting to ready (I/O completes)
  - process terminates
- May be preemptive or non-preemptive

# Scheduling criteria

- To guarantee min/max value, good average, low variance of values such as:
  - CPU utilization (max)
  - Throughput (max)
  - Turnaround time (min)
  - Waiting time (min)
  - Response time (min)
  - other...
- Different algorithms may have different impact on these
- Very workload dependent

# Scheduling algorithms

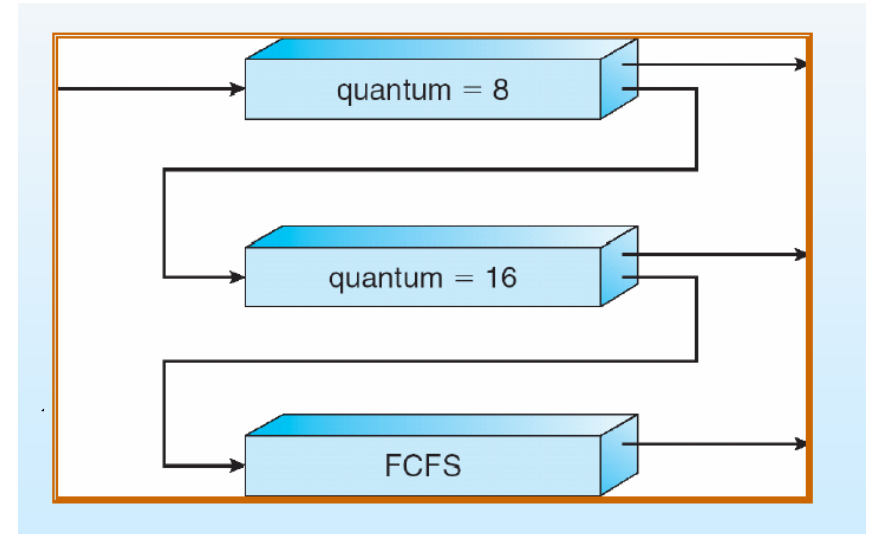
- First Come First Serve
  - non preemptive, can be bad for average waiting time
- Shortest Job First
  - preemptive, heuristic in job duration prediction (e.g., based on prior history...)
- Priority Scheduling
  - introduce priority aging to avoid starvation
- Round Robin
  - like FCFS with preemption, quantum must be large enough wrt context switch time, otherwise overhead too high

- Multilevel Queue

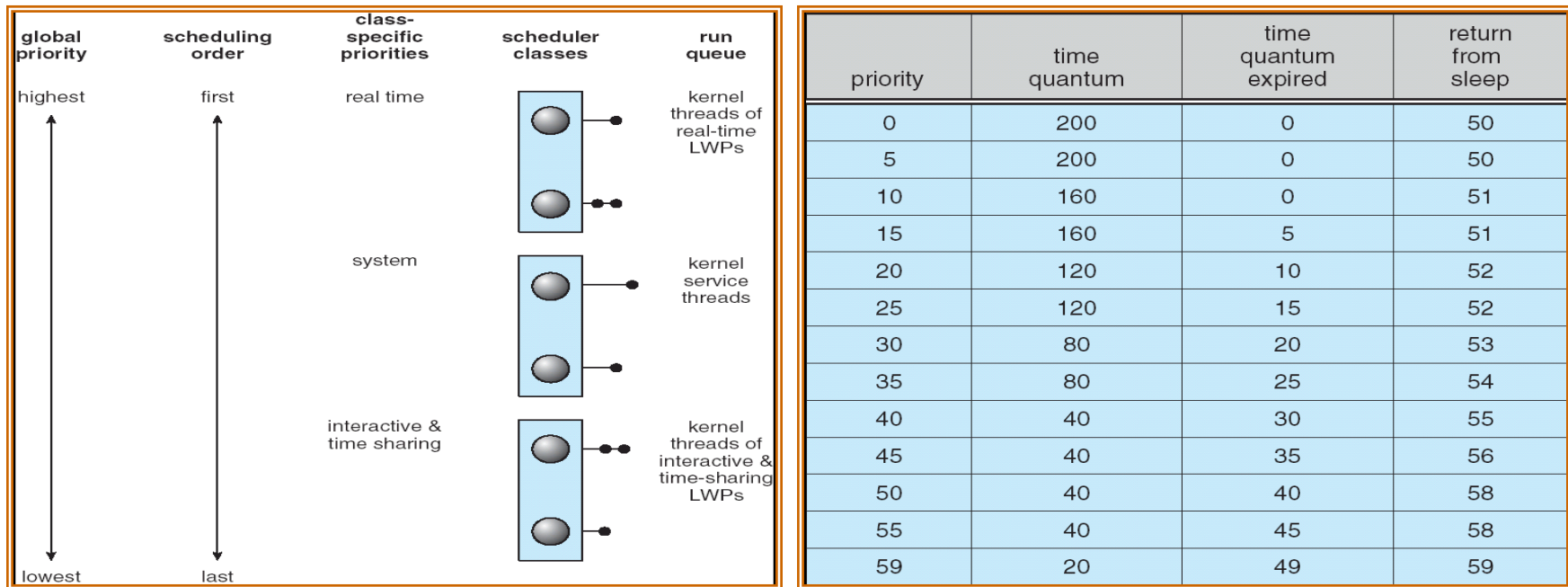
- multiple queues, for different classes of processes (or different priorities)
- within each queue, different scheduling algorithm may be applied
- some objectives – interactive processes should get high priority, compute-bound process may use longer time quanta but non necessarily high priority
- how does the scheduler know which process is what?

# Multilevel Feedback-Queue Scheduling

- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs a service

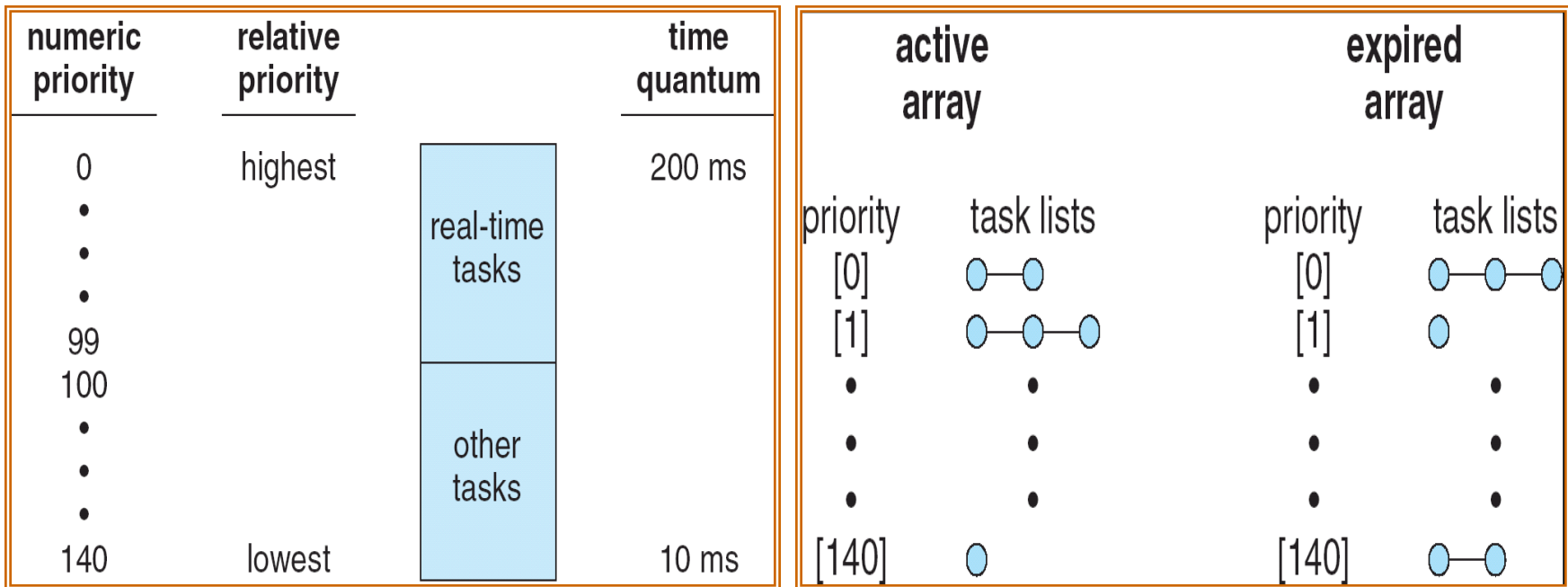


# Example: Solaris



0 = low priority; 59 = high priority  
 high priority task gets smaller quantum

# Example: Linux



higher priority = longer time quantum

for non-RT tasks adjustment criteria sleep time:

long sleeps => interactive => priority - 5

short sleeps => compute intensive => priority + 5

# Scheduling on Multi-processor Systems

- assumption homogeneous
  - in heterogeneous systems, availability of certain resources is a factor (e.g., device, floating point unit, different ISA...)
- objective well balanced systems
  - trade-off between load-balancing, sufficient utilization of all platform components, and processor-affinity
- Per processor runqueue vs. common runqueue
  - Trade-offs
  - Keeping the system balanced
    - pull vs. push

# Scheduling in CMTs

- paper “Chip Multithreading Systems Need a New Operating System Scheduler”
  - emerging platforms today, and all platforms in future will be CMTs
- *Question:* If we have n-way system where each core has m hardware threads, can we deliver performance =  $m \cdot n$  individual cores running in parallel?
  - no but can we get close?
  - objective in paper – design scheduling to maximize performance, by maximizing utilization of CPU resources

# Summary of paper ideas

- improve utilization of processor pipeline (shared resource) in CMT systems
  - idea: consider which units are needed when scheduling next thread, make sure as many of them can be doing work
  - need some easy metric to be able to differentiate between memory- and compute-intensive threads
  - cycles per instruction (CPI) is chosen to be that metric
- Results good for synthetic, but modest for real workloads
  - probably indicates that no reason to put circuitry in hardware which will maintain accurate CPI for the sake of scheduling
- Important for you -> factors which are to be considered
- Next iteration of this work looks at L2 cache utilization, and tries to co-schedule threads which are going to be able to share as much of the cache as possible