System Architecture

14 Scheduling (2)

SMP Scheduling
RT Scheduling
Examples

December 17 2008
Winter Term 2008/09
Gerd Liefländer

SMP Classification

Classification of Multiprocessors

- Loosely coupled multiprocessing
  - each CPU has its own memory and I/O channels, e.g. a cluster of workstations
- Tightly coupled multiprocessing
  - CPUs share main memory
    - ***NUMA*** as well as ***UMA*** systems
  - controlled by one OS
    - *Centralized* or *decentralized scheduling*

Our topic

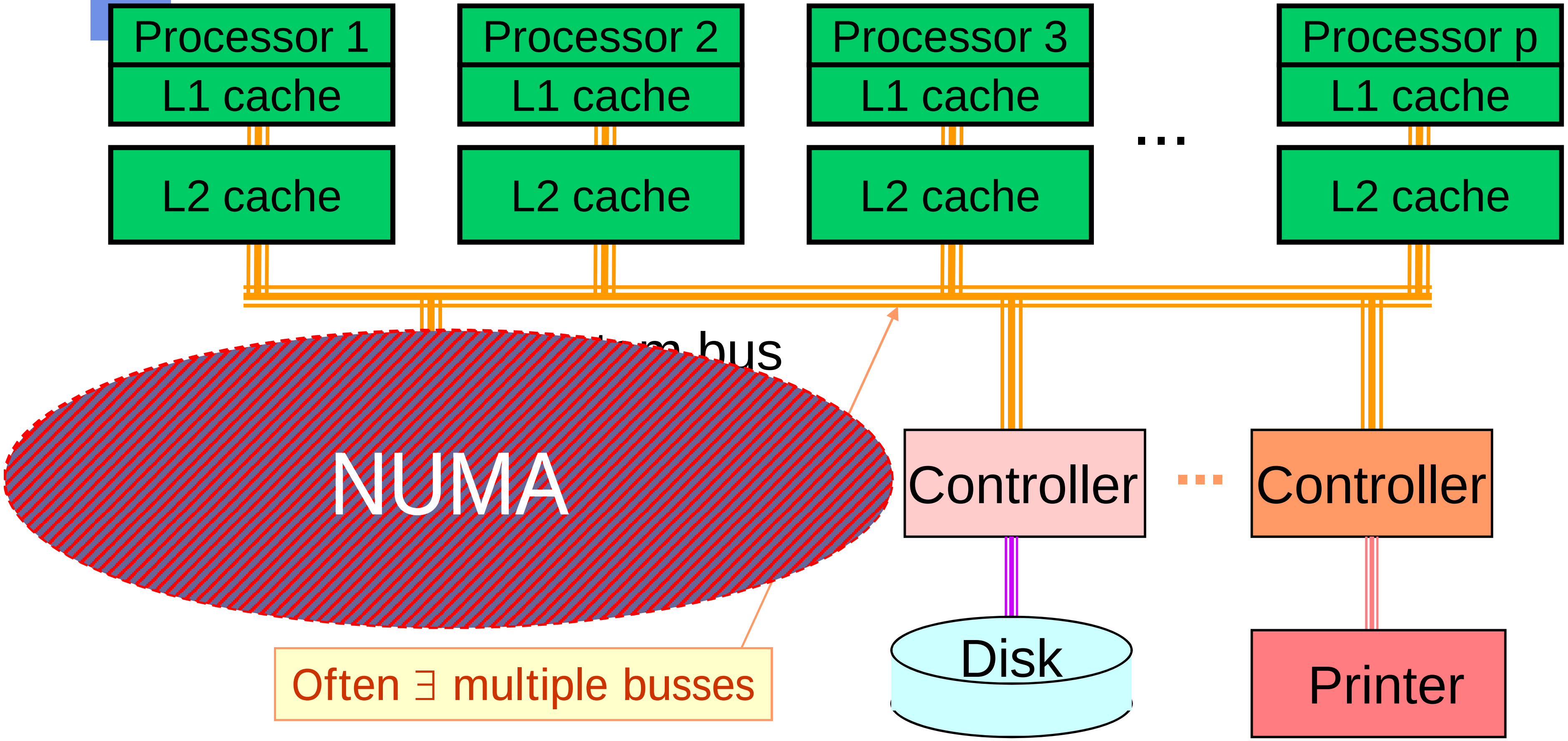
Tightly Coupled Multiprocessors

- Asymmetric Multiprocessing
  - Master/Slave relation
  - Master handles scheduling, interrupt handling etc.
  - Slaves are dedicated to application tasks
  - Main drawback: if master fails ⇒ system fails

- Symmetric Multiprocessing (SMP)
  - Each CPU can handle each task/thread/activity
  - During its life-time a thread can run
    - on any CPU or
    - always on the same CPU (strict processor affinity)
  - Interrupts can be delivered to *each* CPU

Our topic

NUMA Symmetric Multiprocessor



Motivation

Additional Features

Anomalies

Requirements

Motivation

- On a single processor *all CPU related activities* run on the CPU
  ⇒ the only scheduling decision:
  when to run what KLT/Process on the CPU

<u>Simple case:</u>

At one instance of time at most 1 activity is running

- *Are there additional activities that need more attention from the scheduler(s) in a SMP?*
  - Application activities
    - Related processes
    - Threads of a multithreaded application
  - System activities
    - *How to assign a client and its server?*
    - *How to schedule a periodic system activity?*
  - Kernel activities
    - *How to prevent mutual interference of critical paths?*

Motivation

- SMP Scheduling is more complicated because
  - it might pay off that a CPU remains idle even though an application thread is ready
  - ∃ heterogeneous SMPs, i.e. we might have to consider
    - CPUs of different speed or
    - CPUs with different instruction sets
  - ∃ some nice anomalies (see next slide)

SMP Anomalies (R. Graham[1])

- One example with a set p>1 processors and t>1 threads and precedence constraints.
- Graham showed the following anomalies concerning the maximal turnaround time TT_{max}:
  - *Adding* another CPU *increases TT_{max}*
  - *Removing* precedence *constraints* " "
  - *Reducing* the *execution times* " "
  - ...

[1]R.L. Graham: Bounds on Multiprocessor Timing Anomalies, SIAM, J. Applied Mathematics, 1969

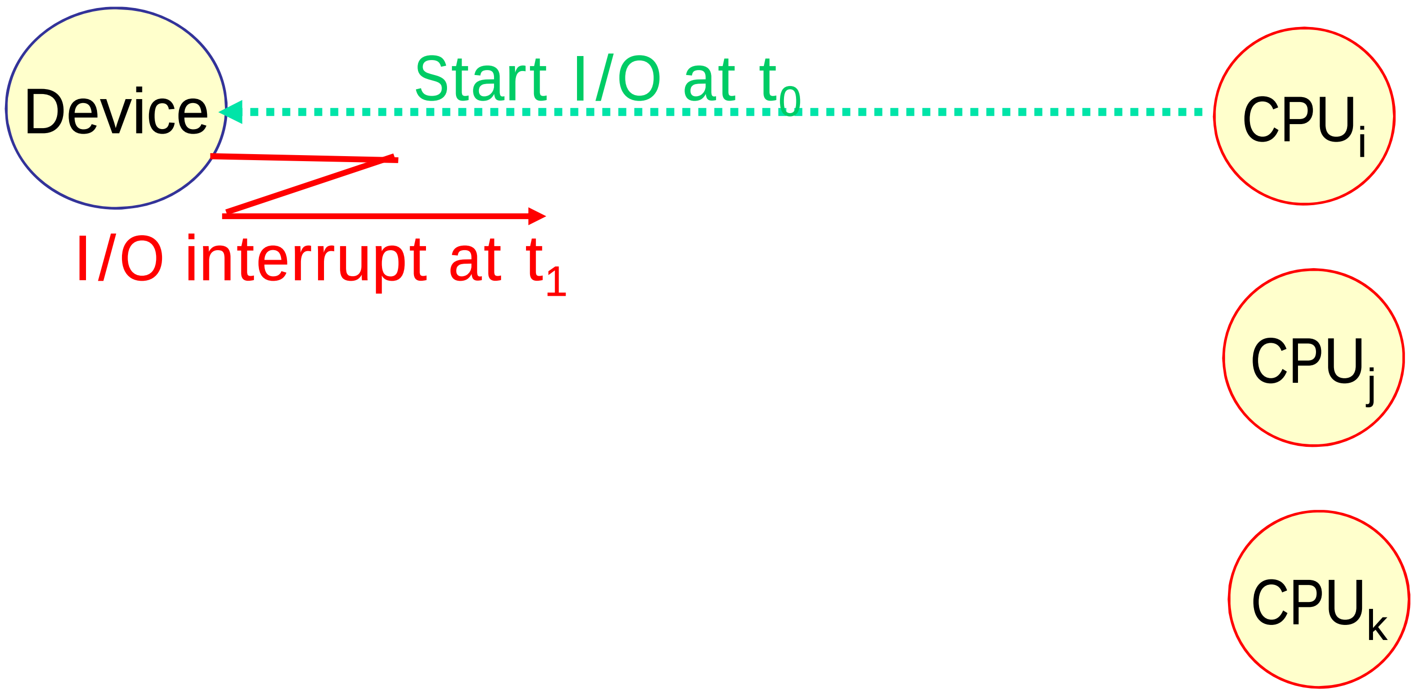
Additional Scheduling Problems

- *Which CPU should handle interrupts?*
- *When creating a new KLT (process) should it run on the same CPU or on another one?*
- *How to schedule threads of a multithreaded task?*
- *Should a thread (process) stay on its first CPU or can it run on another one?*
- *What are useful scheduling criteria for migrating or pinning threads?*
- *When to get this scheduling information and how to collect and store it?*

See diploma thesis and later work of Jan Stöß

Handling Interrupts

Handling Interrupts on an SMP



Scheduling Interrupt Handlers (1)

Five different scheduling policies:

0. Device is dedicated to a specific processor

1. Interrupt can be handled on every processor

2. Interrupt "should" be handled on the processor having initiated previous I/O-activity, because
   - thread having initiated I/O is *bound to this processor* (see *affinity* in WinXP ...) and still has some *cache footprint* on this processor
   - thread is still running on that CPU in case of a previous asynchronous I/O ($\Rightarrow$ no thread switch)

Scheduling Interrupts (2)

3. Interrupt can be handled on a CPU that is currently executing another interrupt handler
   - can save one mode switch from user $\rightarrow$ kernel
   - can postpone interrupt handling due to interrupt convoys (and furthermore the innocent current thread is punished multiple times)

4. Interrupt can be handled on the processor with "lowest priority activity" (i.e. idle thread)

Handling New Threads

Running a New Thread

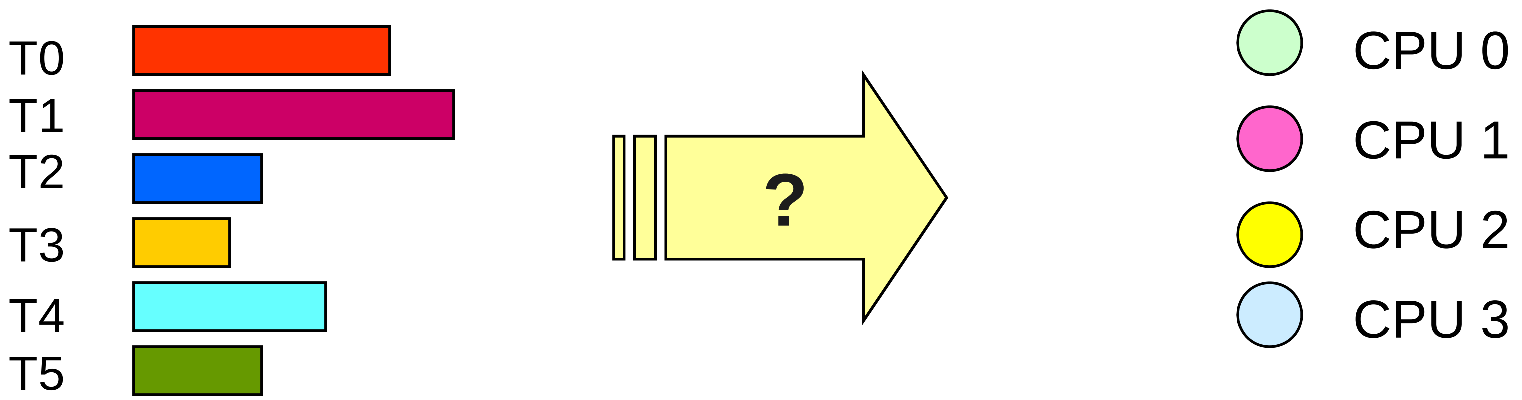
- Depending on the application it could be better to schedule the new thread
  - on the CPU that has created this thread
    - because the creating thread and the creator might cooperate on common shared data (e.g. KLTs of the same task)
  - on a specific CPU
    - A nearby CPU to improve collaboration via a shared L3-cache or on a NUMA
    - To balance the load of system or of the application
    - The application programmer knows about the special features of the CPU (in a heterogeneous SMP)
  - on any other CPU just to improve parallelism

Scheduling Threads and Tasks

- Single-threaded tasks, i.e. processes
  - scheduling processes sharing code or data to the same processor can reduce
    - cache loading time
    - TLB loading time
  - anonymous scheduling of processes can reduce turnaround times
- Multi-threaded tasks
  - scheduling all threads of one task to the same CPU can save
    - cache + TLB loading time in case you can switch within the AS,
    - but also reduces concurrency completely
  - scheduling threads of a task on as many CPUs as possible supports concurrency, but may lengthen cache loading time
  - scheduling threads of one task at the same time (gang-scheduling) can profit from their parallel execution

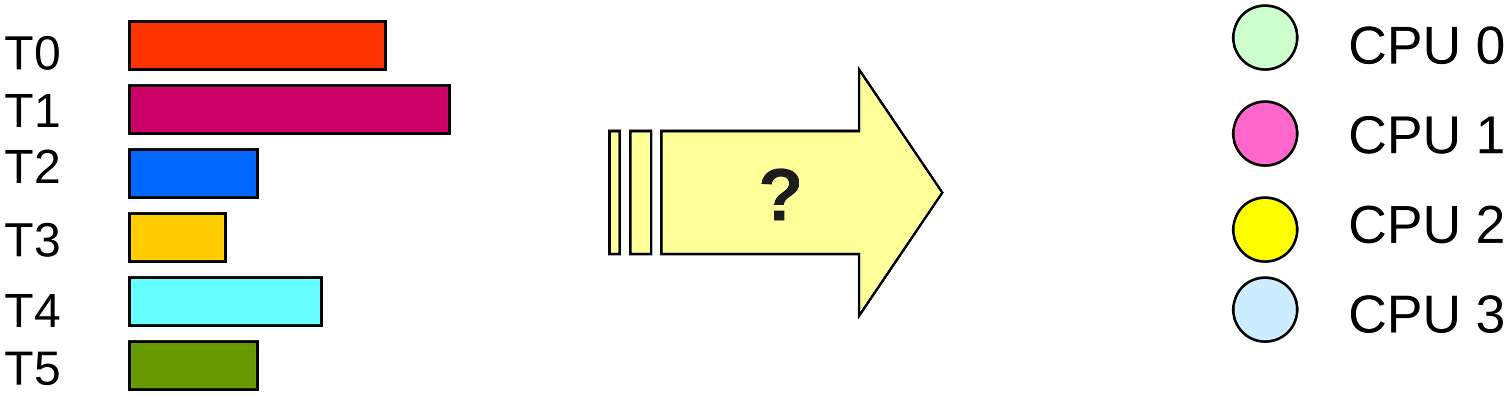
Additional Scheduling Parameters

Suppose, you have to schedule the following multi-threaded application on an empty, tightly-coupled 4-processor multi-programming system.



1. Number of processors to be involved
2. Precedence relation
3. Communication costs

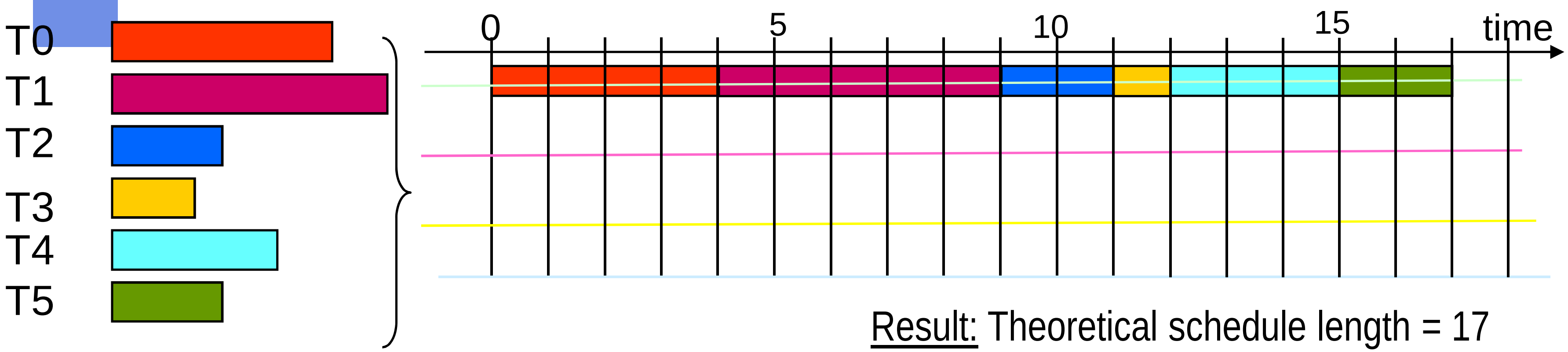
Additional Scheduling Parameters



1. Scheduling parameter: Number of processors to be involved
   - 1 processor
   - p' < p processors
   - all p processors

Discuss Pros and Cons of each of the above possibilities.

Additional Scheduling Parameters



1. Scheduling parameter: Number of processors to be involved:

1 processor (suppose CPU 0):

Pro: Identical to a solution on a single-processor system
Unused processors may be reserved for other applications

Con: You do not use the offered parallelism of the hardware
thus your turnaround time is high.

Additional Scheduling Parameters

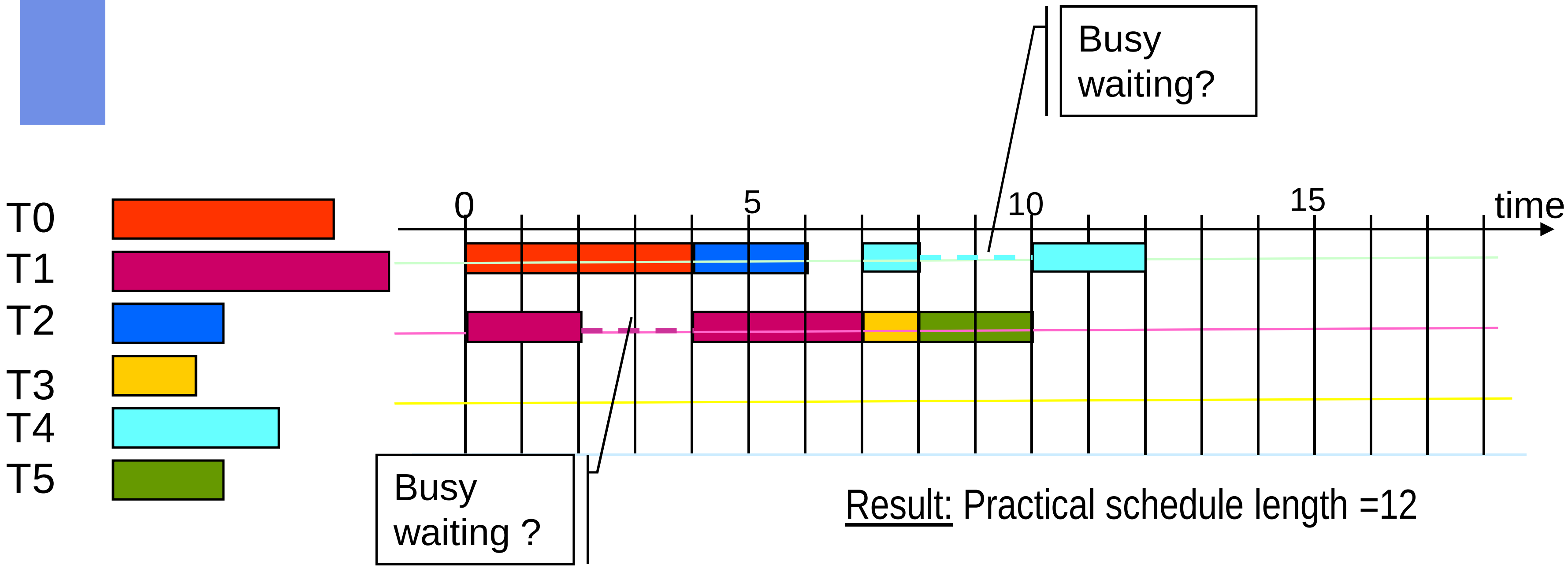
Result: Theoretical schedule length = 9

1. Scheduling parameter: Number of processors to be involved:
   p' < p processors (suppose p' = 2, CPU 0 and CPU 1):

Pro: Theoretically smaller maximal turnaround time due to the parallel execution of T_j

Con: Due to critical sections within these T_j the individual turnaround times of the T_j may be larger

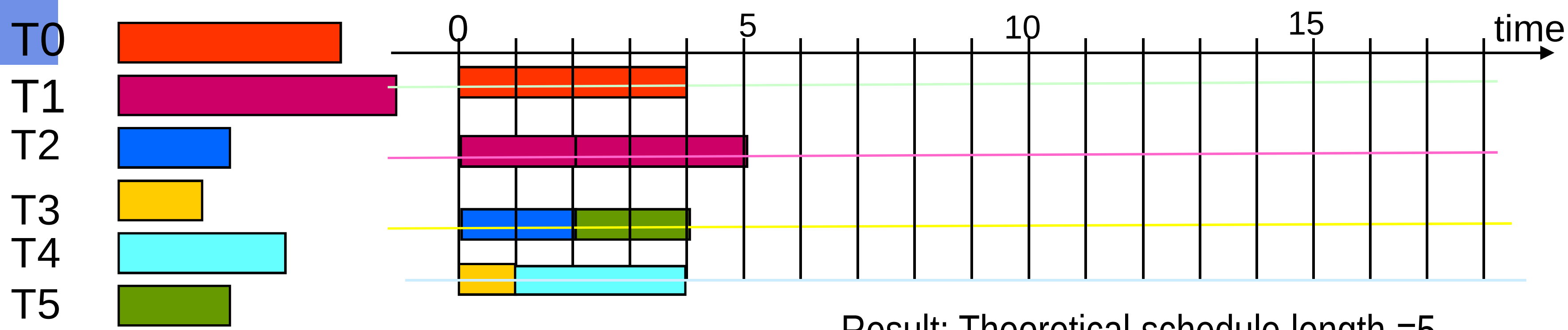
Additional Scheduling Parameters



1. Scheduling parameter: Number of processors to be involved
   p’ < p processors (suppose CPU 0 and CPU 1):

Are there further constraints leading to a longer schedule?

Additional Scheduling Parameters



Result: Theoretical schedule length =5

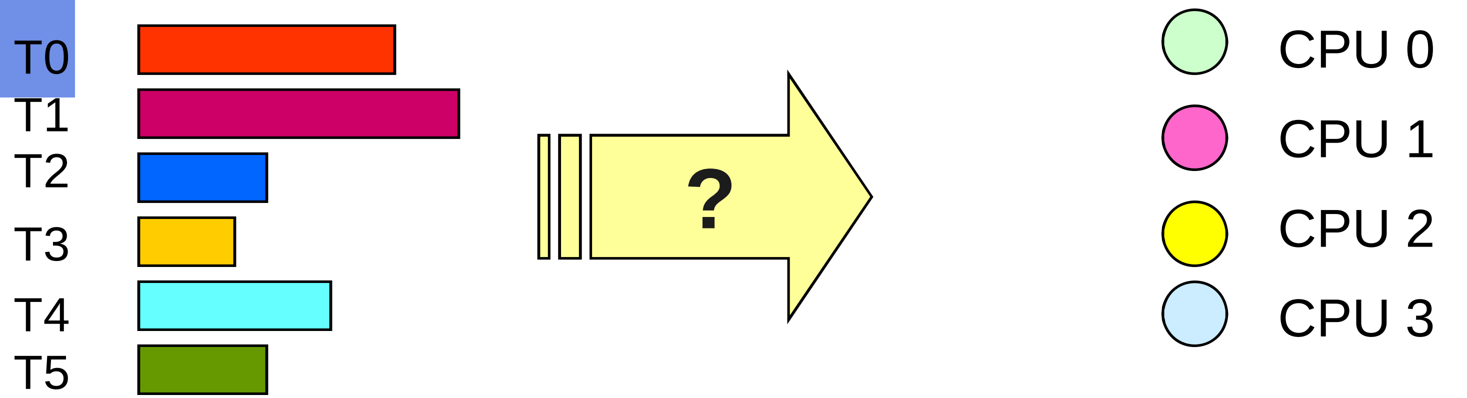
1. Scheduling parameter: Number of processors to be involved

   p processors

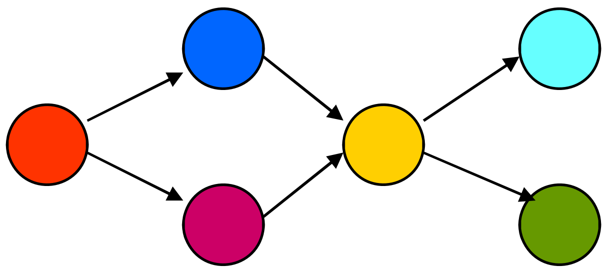
Pro: Theoretically shortest maximal turnaround time due to the parallel execution of T_j

Con: Due to critical sections within these T_j the individual turnaround times may be larger

Additional Scheduling Parameters

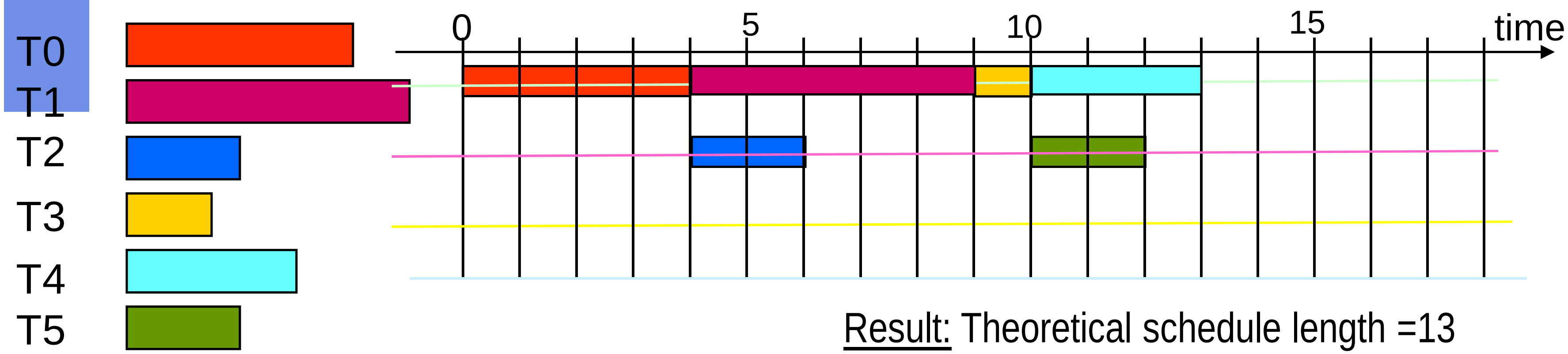


2. Scheduling parameter: precedence constraints, i.e. a certain Ti has to be finished before Tj may start to execute



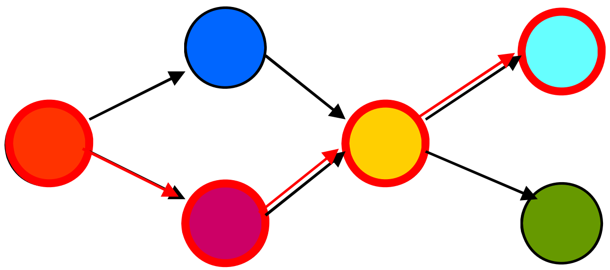
Arrows indicate the precedence relation

Additional Scheduling Parameters

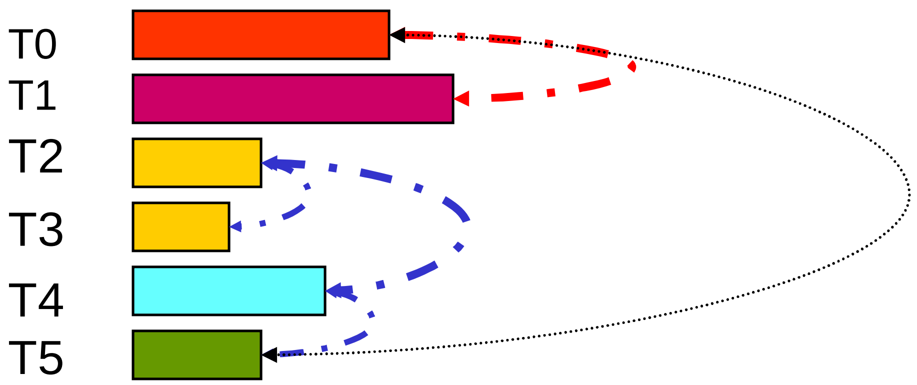


Result: Theoretical schedule length =13

How to solve this problem?
From the graph we know that at most two CPUs are required $\Rightarrow$
Determine the critical path
Assign to CPU0 according to precedence constraints
Assign to CPU1 whenever possible



Additional Scheduling Parameters



3. Scheduling parameter: Communication costs between threads

Communication between threads on different processors has to be done via main memory. Communication between threads on the same processor could be done via caches or registers.

Conclusion:

What you might gain by real parallelism, you might lose due to increased communication costs. $\Rightarrow$ careful analysis is required

SMP Scheduling Policies

Scheduling of Multithreaded Tasks

- Anonymous (dynamic) thread scheduling
  - assign KLT/process to next available CPU ⇒ threads will run on different CPUs during their life cycle
- Dedicated scheduling
  - threads of the same application are assigned to a specific processor-subset (processor-affinity)
- Adaptive scheduling
  - Scheduler maps threads (statically or dynamically) according to load etc.
- User based scheduling
  - User (programmer) can pin threads temporarily to dedicated processors

Scheduling Tasks

- Processes (Single threaded tasks)
  - Fair Share Scheduling between processes = straightforward
  - Prefer tasks with partly shared address space (swap in/out)
- Multi threaded task
  - Fair Share Scheduling on task or thread basis
  - Swap in/out complete task

Remark:
The VAX VMS OS supports another scheduling unit:
"session" which is directly related to a user, thus
you can establish fair share scheduling on session basis.

Gang Scheduling

- Simultaneous scheduling of threads of a task
- Useful for applications where performance severely degrades when one part of the task is not running
- Threads often need to synchronize with each other, e.g. after another iteration step for the solution of a difference equation (e.g. at a barrier synchronization)

SMP Ready Queue(s)

Centralized ↔ Decentralized Sched.

- Centralized SMP scheduling with
  - 1 global ready queue

    Pro: easy to implement a consistent policy

    Con: not that scalable for m >> 1 CPUs

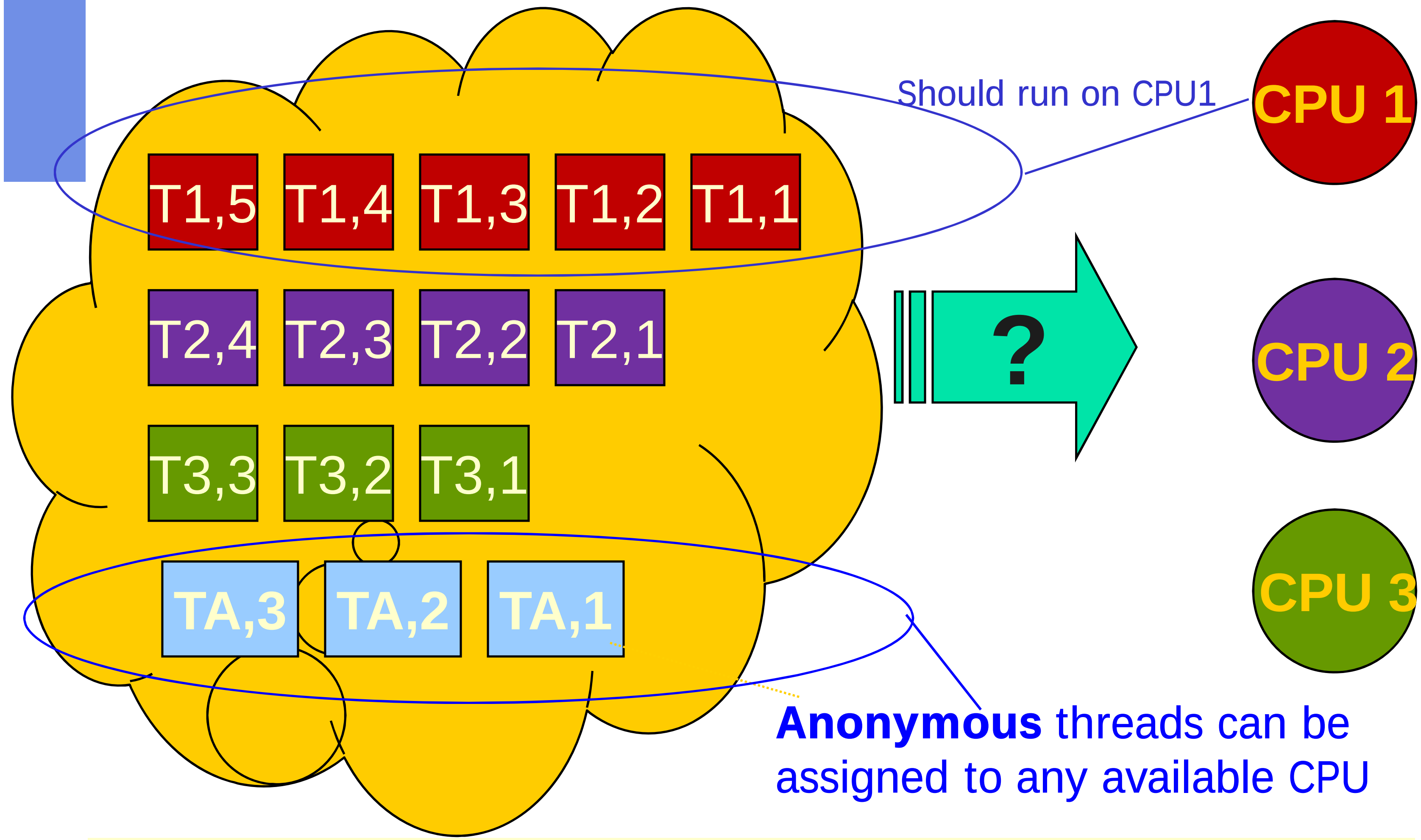
- Decentralized scheduling with
  - n>1 local ready queues

    Pro: fewer access conflicts at each “local” ready queue

    Con: ∃ load balancing problem

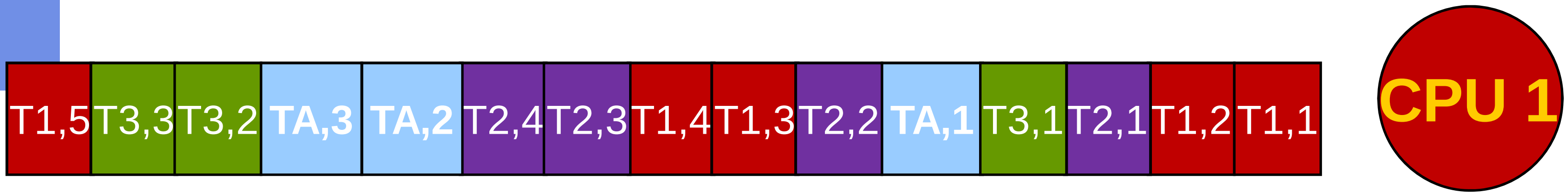
  - *Problem: How to fill ready queues when new threads arrive (or come back after some waiting period)?*

Dedicated & Anonymous Threads



Find an efficient data structure for the ready queue(s)

Randomly Ordered Ready Queue

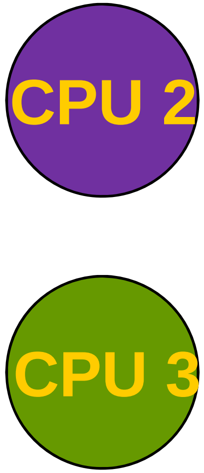


Policy: Assign “first fitting thread”

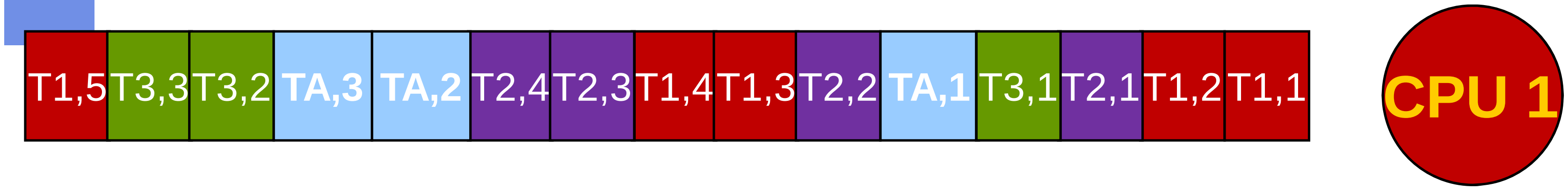
Drawbacks:

1. You do not assign head of the ready queue, ⇒ there is some additional overhead for looking up

2. You might assign an anonymous thread to CPUx, even though there is a dedicated thread Tx for CPUx. Thus, one of the other CPUs can be idle next!



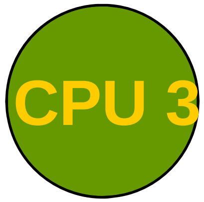
Randomly Ordered Ready Queue



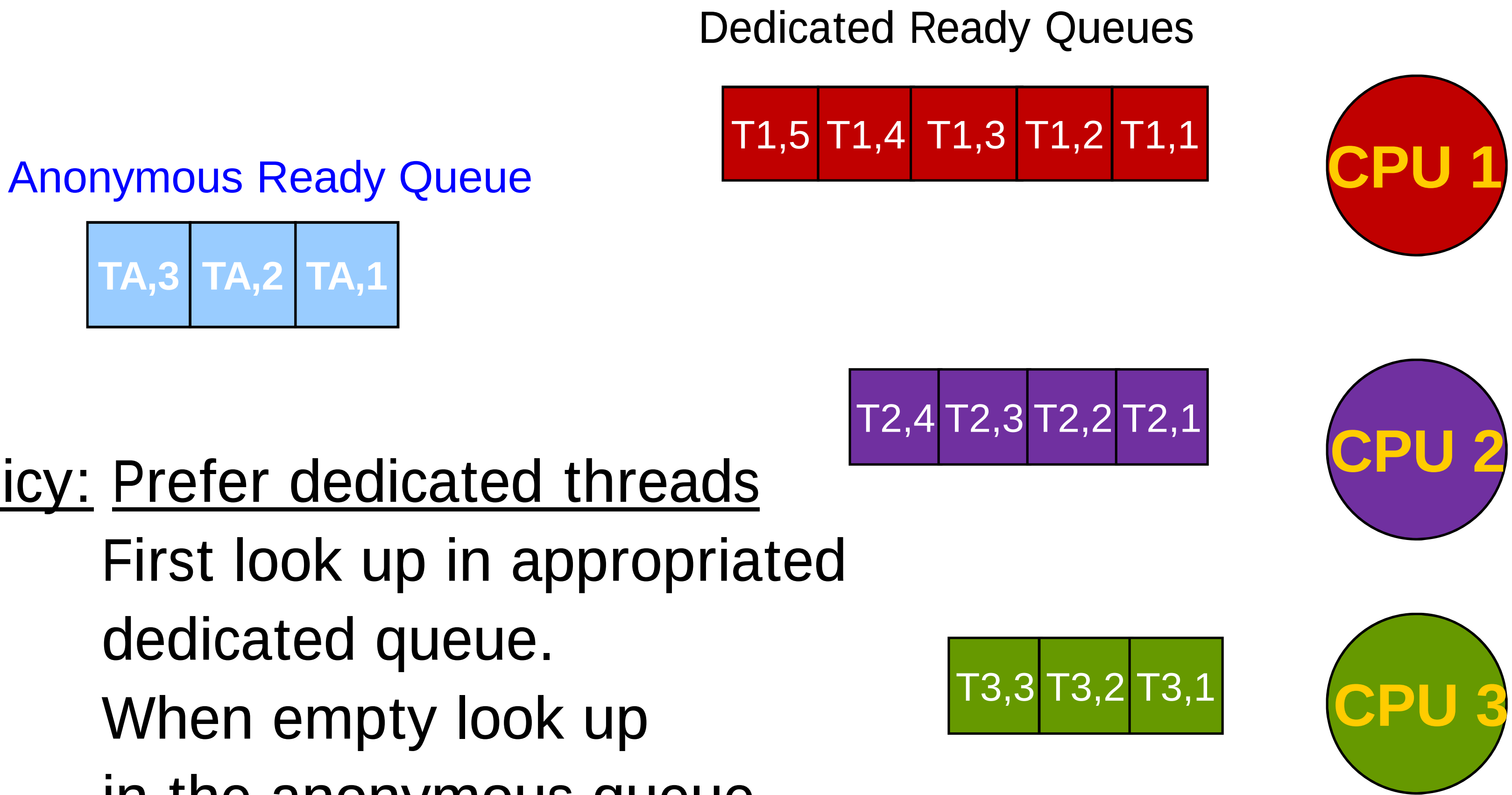
Policy: Assign "best fitting thread"

Drawback:

You may have to look through the entire ready queue, i.e. O(n)-scheduler



Anonym. & Dedic. Ready Queues



<u>Policy:</u> <u>Prefer dedicated threads</u>

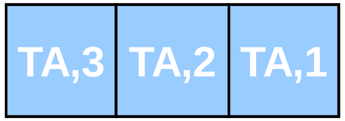
First look up in appropriated dedicated queue.

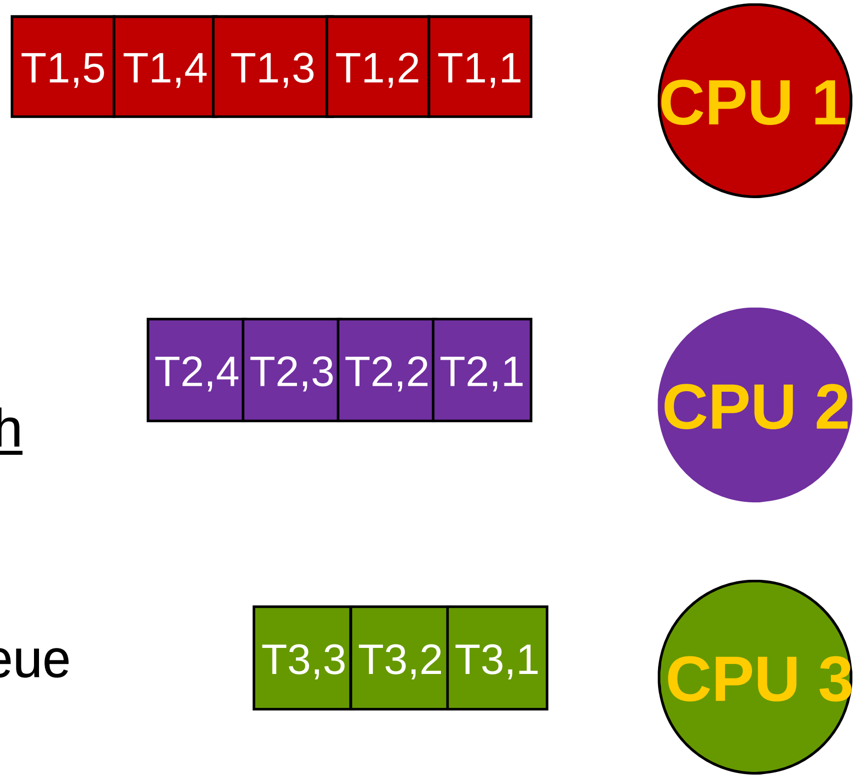
When empty look up in the anonymous queue

Anonym. & Dedic. Ready Queues

Dedicated Ready Queues

Anonymous Ready Queue





<u>Policy:</u> <u>Strictly prefer threads with higher priority</u>

Compare the head of the appropriate dedicated queue with the head of the anonymous queue

Pick the one with the higher priority

Real-Time Scheduling

Real-Time Scheduling

- Correctness of the system may depend
  - not only on the logical result of the computation
  - but also on the time when these results are produced, e.g.
  - tasks attempt to control events or to react to events that take place in the outside world
- These external events occur in “real time” and processing must be able to keep up with them
- Processing must happen in a timely fashion, neither too late, nor too early.

Real Time System (RTS)

- RTS accepts an activity A and guarantees its requested (timely) behavior B if and only if

  RTS finds a schedule that
  - includes all already accepted activities A_i and the new activity A,
  - guarantees all requested timely behaviors B_i and B, and
  - can be enforced by the RTS.
- Otherwise, RT system rejects the new activity A.

Typical Real Time Systems

- Control of laboratory experiments
- Robotics
- (Air) Traffic control
- Controlling Cars / Trains/ Planes
- Telecommunications
- Medical support (Remote Surgery, Emergency room)
- Multi-Media ...

Remark:
Some applications may have only soft real-time requirements, but some have really hard real-time requirements

Hard Real-Time Systems

Requirements:

Must always meet all deadlines (time guarantees)

You must guarantee that these applications are done in time, otherwise a catastrophe might happen

Examples:

1. If the automatic landing of a jet cannot react to sudden side-winds within some ms a severe crash might occur.
2. An airbag system or the ABS has to react within some ms
3. Remote scalpel in a surgical operation must immediately follow all movements of the surgeon

Soft Real-Time Systems

Requirements:

Must mostly meet all deadlines, e.g. in 99.9%

Examples:

- Multimedia: 100 frames per day might be dropped (late)
- Car navigation: 5 late announcements per week are acceptable
- Washing machine: washing 10 sec over time might occur once in 10 runs, 50 sec once in 1000 runs.

Examples of Scheduling

Linux 2.4 Scheduling

- Linux offers three scheduling policies
  - A traditional scheduler **SCHED_OTHER**[1]
  - Two soft-real-time scheduler (mandated by Posix.1b)
    - **SCHED_FIFO**
    - **SCHED_RR**
  - They give the CPU to a real-time process whenever such a real-time process become ready (except when already a real-time process is executing)

[1]These three scheduling policies are an attribute of the TCB

Priorities

- Static “priority”
  - Maximum size of the time slice a process should be allowed before being forced to allow other processes to compete for the CPU
- Dynamic priority
  - Amount of time remaining in this time slice; declines with time as long as the process runs on the CPU
  - When its dynamic priority is 0, the process is marked for rescheduling
- Real-time priority
  - Only real-time processes can get the real-time priority values
  - Higher-real-time priority values always beat lower priorities, i.e. preempt the corresponding process

Related Entries in the TCB

`long counter`	Time remaining in the process's current quantum (~dyn. priority)
`long nice`	process's nice value, -20 … +19 (~ static prio)
`unsigned long policy`	**SCHED_OTHER, …_FIFO. …_RR**
`struct mm struct *mm`	points to the memory descriptor
`int processor`	CPU ID on which process will run
`unsigned long cpus_runnable`	
`unsigned long cpus_allowed`	CPUs allowed to run
`struct list_head run_list`	head of the run_queue
`unsigned long rt_priority`	real-time priority

Linux 2.4 Real-Time Scheduling

- SCHED_FIFO
  - The corresponding real-time process runs until it either blocks on I/O, blocks to another waiting event, explicitly yields the CPU, or is preempted by another real-time process with a higher real-time priority
  - Acts as it has an unbounded time-slice
- SCHED_RR
  - As above except that time-slice matters, i.e. when a SXCHED_RR process's time-slices expires, its PCB is appended to the corresponding run-sub-queue to give other SCHED_RR processes with the same priority the chance to run instead of

Linux 2.4 Scheduling Quanta

- Linux gets a timer interrupt (or tick) once every 10 ms on a IA-32
  - An ALPHA port of the Linux kernel issues 1024 ticks per second
- Linux wants the time slice to be ~ 50 ms

Linux 2.4 Epochs

- Linux scheduling works by dividing the CPU time into epochs
  - In a single epoch, every process has a specific time quantum whose duration is computed when the epoch begins
  - The epoch ends when all runnable processes have exhausted their time quanta
  - The scheduler recomputes the time-quanta of all processes and a new epoch begins
- The base time quantum of a process is computed according to its nice value

Selecting the next Process to run

```
repeat_schedule:
   next = idel_task(this_cpu);
   c = -1000;
  list_for_each(tmp,&runqueue_head({
        p = list_entry(tmp, struct task_struct, run_list);
        if (can_schedule(p, this_cpu)){
             int weight = goodness(p, this_cpu,
                        prev->active_mm);
              if (weight >c) c = weight, next =p;
        }
   }
```

Recalculating Counters

```
if(unlikely(!c) {  /new epoch begins …*/
    struct task_struct *p;
    spin_unlock_irq(&runqueue_lock);
read_lock(&tasklist_lock);
for_each_task(p)
  p->counter =(p->counter >>1)+
               NICE_TO_TICKS(p->nice);
 read_unlock(&tasklist_lock);
 spin_lock_irq(&runqueue_lock);
 goto repeat_schedule);
 }
```

Calculating goodness()

```
static inline int goodness(p, this_cpu, this_mm){
  int weight = -1;
  if (p->policy == SCHED_OTHER){
     weight =p->counter;
     if (!weight) goto out;
     if (p->mm == this_mm || !p->mm)
       weight += 1;
     weight +=20 – p->nice;
     goto out;
  }
  weight = 1000 + p->rt_priotity;
out: return weight;
```

`weight = 0` i.e. p has exhausted its quantum

`0 < weight < 1000`
P is a conventional process

`weight >= 1000`
p is a real-time process

Linux 2.4 Scheduler NOT Scalable

- The run_queue is protected by one run queue lock
  - As the number of CPUs increases, lock contention also increases
  - It is expensive to recalculate goodness() for every process on every invocation of the scheduler
    - A profile of the Linux 2.4 kernel during the VolanoMark benchmark runs showed that 37-55% of total time spent in the kernel is spent in the scheduler
    - VolanoMark benchmark establishes a socket connection to a chat server for each simulated chat room user. For a 5 to 25-chatroom simulation, the kernel must potentially live with 200 to 400 threads.

Linux 2.4 Scheduling Quanta

- with epochs and time-quanta
- The life of a process is subdivided into epochs
- Time-quanta are dependent on the processes and their epochs
  - Each process has a time-quanta base, i.e. nice value
    - 20 ticks ~ 210 ms
    - Time-quanta decreases periodically with every tick
    - **`quantum = quantum/2 +(20-nice)/4 +1`**

Linux 2.4

- Kernel distinguishes between 3 scheduling policies
  - FCFS for preemptable, cooperative real-time processes
  - RR for time-sliced real-time processes
  - Priority based (+RR) for all other time-shared processes
- Process selection depends on a quality function c in O(n):

`c = -1000`	if process is the **`init`** process
`c = 0`	if process has expired its time quantum
`0 < c < 1000`	if process has not expired its time quantum
c ≥ 1000	if process is a real-time process

- Processes (KLTs) can get a bonus (boost) if they share an AS with the process (KLT) that has been executing before on the same CPU

<u>Hint:</u>

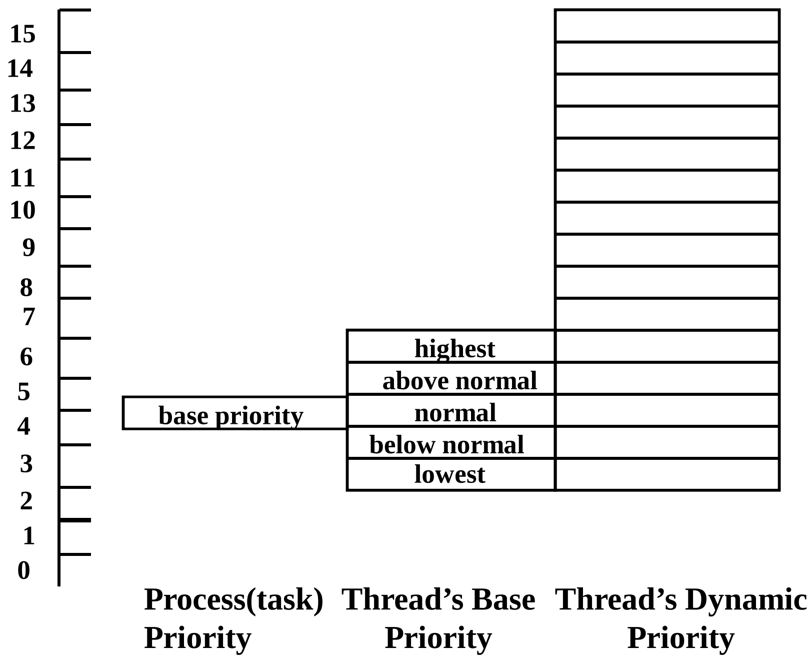
Study slides of the corresponding talks of previous Proseminars “Linux Internals”

Linux 2.5 Scheduler with O(1)

- Any scheduling is done with **constant complexity**
- There are two tables per processor: active and expired
- Priorities:
  - 1 – 100 for real-time processes
  - 101 – 140 for best-effort processes
  - Per priority level a double linked list per table
  - Priorities of best effort processes depend on degree of their interactivity, i.e.
    - bonus = -5 for interactive and bonus = +5 for compute bound
    - New calculation at the end of a time-slice, i.e.: prio = MAX_RT_PRIO + nice + 20 + bonus
  - Having expired its time quantum the PCB is handed over to the corresponding expired table
  - If no element is left over in the active-table the role of both tables is switched, i.e. we have a change of epoch tables

Windows NT Priority

Windows supports fixed priorities [16, 31] for real-time applications. Time-shared applications may change their priorities within [0,15] according to their behavior concerning I/O bursts and CPU bursts.



If n>2 processors are available, (n-1) are busy with the (n-1) highest priority threads whereas the remaining processor executes all remaining ready threads.
You have the ability to pin a task or its threads to specific processors.

UNIX SV"R4" Scheduling

Set of 160 priority levels divided into three priority classes
Because basic kernel is not preemptive some spots called preemption points have been added, allowing better reaction times for real-time applications

Priority Class	Global Value	Scheduling Sequence
Real-time	159 • • • • 100	first
Kernel	99 • • 60	
Time-shared	59 • • • • 0	last

A dispatching queue per priority is implemented, processes on the same priority level are executed in RR.

Real-time processes have fixed priorities (and fixed time slices,) time-shared processes have dynamic priorities and varying time slices in the range **[10, 100]** ms.

Unix 4.3 BSD

- MLFQ 32 ready queues, each per RR + dynamic priorities $\in$ [0,127]
- *How to determine the dynamic priorities* **`p_usrpri`**?
- After each 4th tick (~ 40 ms)
  - **`p_usrpri = PUSER+[ p_cpu/4]+2*p_nice`**
    - with **`p_cpu = p_cpu +1`** with each tick (10 ms)
    - with weigth: **`-20`** $\leq$ **`p_nice`** $\leq$ **`20`**
- Smoothing of CPU utilization **`p_cpu`** per s
  - **`p_cpu = 2*load/(2*load+1)*p_cpu+p_nice`**
  - however, processes with sleep-time > 1 s

    $p\_cpu = (2*load/(2*load+1))^{p\_slptime} * p\_cpu$

Unix 4.3 BSD Example

- Assumption 1: average load = 1 ⇒
  `p_cpu = (2*1)/(2*1+1)*P_cpu + p_nice`
  `= 0.66*p_cpu + p_nice`
- Assumption2: process collects T_i ticks in time interval i
- Assumption 3: `p_nice = 0`
- $$\begin{aligned} p\_cpu &= 0.66*T_0 \\ &= 0.66*/T_1+0.66*T_0)= 0.66*T_1 + 0.44*T_0 \\ &= 0.66*T_2 + 0.44*T_1+ 0.3*T_0 \\ &=0.66*T_3 + \ldots + 0.20*T_0 \\ &=0.66*T_4 + \ldots +0.13*T_0 \end{aligned}$$

⇒ After 5 s only 13% of the primary CPU load are counted to get a new `p_cpu value`

Summary: Scheduling Parameters

- Priority
  - Static versus dynamic priority
  - How to distinguish between I/O-bound and compute bound processes?
  - When and how much to increase the priority of an I/O-bound process
  - How to perform aging to prevent starving
- Time slice (quantum)
  - I/O bound processes do not need long time slices
  - CPU bound processes prefer long time slices
  - Short time slice: switching overhead
  - Long time slices: poor response time for interactive processes?
  - Higher priority means longer time slices?

Recommended Reading

- J. Apoovo et al.: "Scheduling in K 42"
- Bacon, J.: Operating Systems (6)
- Nehmer, J.: Grundlagen moderner BS (5)
- Stallings, W.: Operating Systems (9,10)
- Silberschatz, A.: Operating System Concepts (5)
- Tanenbaum, A.: Modern Operating Systems (2, 8)
- Wettstein, H.: Systemarchitektur (9)
- Stöß, J.: Using OS Instrumentation and Event Logging to Support User-Level Multi-processor Schedulers, Diplomarbeit 2005
- Stöß, J. et. Al.: Flexible, Low-Overhead Logging to support Resource Scheduling, ICPADS 06, Minneapolis, July 2006

Recommended Reading

- Nussbaum, D. et al.: Chip Multithreading Systems need a New OS Scheduler”, SIGOPS, 2004
- T. Anderson et al.: The Performance Implications of Thread Management Alternatives for SMP”, SIGMETRICS, 1989
- T. Anderson et al.: “*Scheduler Activations*: Effective *Kernel Support* for the User-Level Management of Parallelism”, SOSP, 1991
- J. Barton et al.: “A Scalable Multi-Discipline Multiple-Processor Scheduling Framework for *IRIX*”, IPPS, 1995

Recommended Reading

- F. Bellosa: “Follow-on Scheduling: Using TLB Information to Reduce Cache Misses”, SOSP 1997
- F. Bellosa et al.: The Performance Implications of Locality Information Usage in SMPs”, Journal of Parallel and Distrnbuted Computing, 1996
- D. Black: “*Scheduling Support* for Concurrency and Parallelism in the *Mach OS*”, IEEE Computer, 1990
- Casavant,T.L.; Kuhl,J.G.: A Taxonomy of Scheduling in *General-Purpose Distribute Computing Systems*”, IEEE Trans. O. Software Engineering, 1998
- G. Feitelson. "Job Scheduling in Multiprogrammed Parallel Systems". IBM Research Report RC 1979.
- H. Franke et al.: PMQS: *Scalable Linux Scheduling* for High End Servers”, 5. Annual Linux Showcase and Conference, 2001

Additional Reading

- D. Ghosal et al.: “The Processor Working Set and its Use in Scheduling SMPs”, IEEE Transactions on Software Engineering, 1991
- A. Gupta et al.: The Impact of OS Scheduling Policies and Synchronization Methods of Performance on Parallel Applications”, SIGMETRICS, 1991
- L. Kontothanassis te al.: Scheduler-Conscious Synchronization”, ACM Transactions on Computer Systems, 1995
- S. Leutenegger et al.: The Performance of Multiprogrammed SMP Scheduling Policies”, SIGMETRIVCS, 1990
- B. Marsh et al.: First-Class User Level Threads, SOSP, 1991

Recommended Reading

- J. Ousterhout: “Scheduling Techniques for Concurrent Systems”, ICDCS, 1982
- K. Sevcik: “Characterizations of Parallelism in Applications and their Use in Scheduling”, SIGMETRICS, 1989
- M. Squillante et al.: “Using Processor Cache Affinity Information in SMP Scheduling”, IEEE on Parallel and Distr. Systems, 1993
- A. Tucker et al.: “Process Control and Scheduling Issues for Multiprogrammed SMPs”, SOSP, 1989
- S. Zhou et al.: “Processor-Pool Based Scheduling for Large-Scale Numa Multiprocessors”, SIGMETRCS, 1991

Agenda

- Classification of SMPs
- Motivation
- Handling Interrupts
- Handling New Threads
- SMP Scheduling Policies
- SMP Ready Queues
- Real-Time Scheduling — Not examined
- Examples
  - Linux, Unix, Windows
  - User Level Scheduling of L4-SMPs — Not examined