System Architecture

7 Thread States, Dispatching

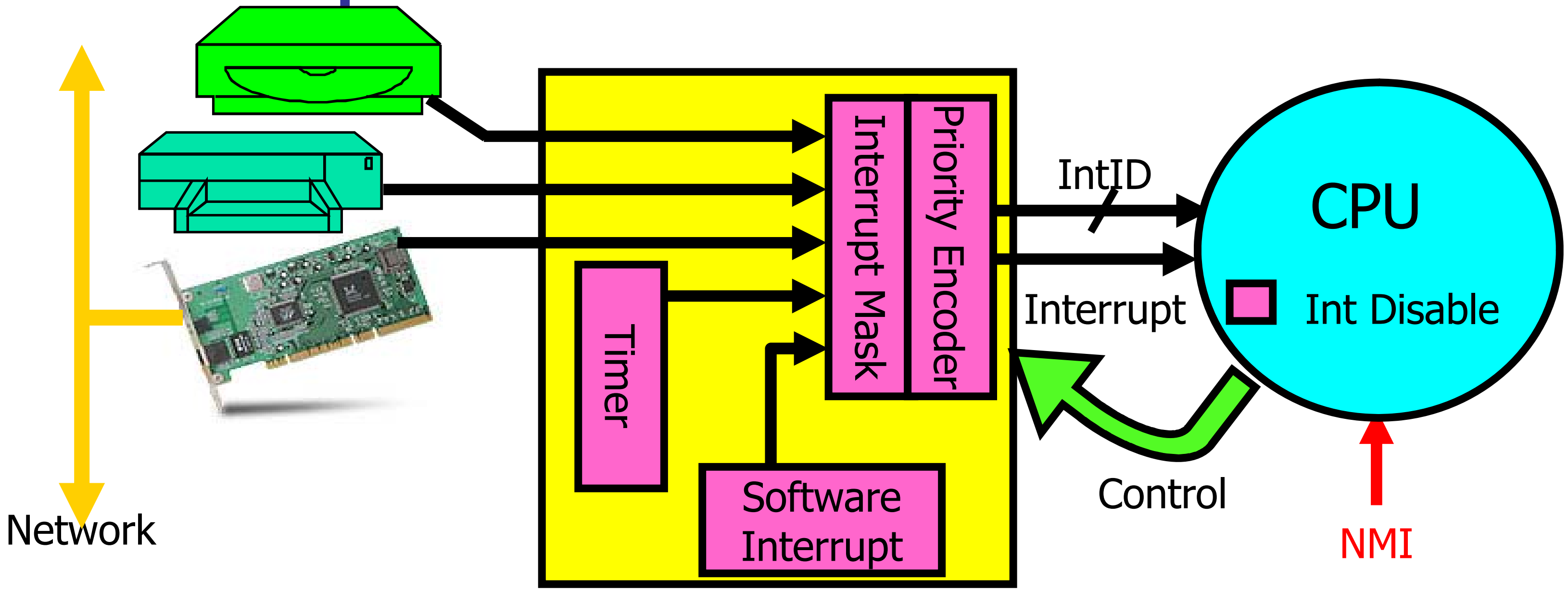
Thread States, Dispatching,
Cooperating Threads

November 17 2008
Winter Term 2008/09
Gerd Liefländer

Agenda

- Review: Interrupts, Activity Switches
- Motivation
- Thread State Models
- Implementing Thread States
- Consequences for Dispatching
- Relation between Task & Thread States
- Cooperating Threads

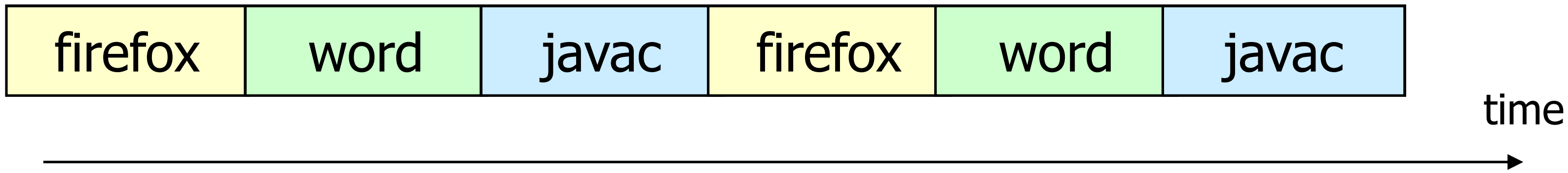
Interrupt Controller



- Interrupts invoked with interrupt lines from devices
- Interrupt controller chooses interrupt request to honor
  - Mask enables/disables interrupts
  - Priority encoder picks highest enabled interrupt
  - Software Interrupt Set/Cleared by Software
  - Interrupt identity specified with ID line
- CPU can disable some interrupts with internal flag
- Non maskable interrupt (NMI) can not be disabled

Multiprogramming

- Running multiple <u>applications</u> concurrently
- Requires multiplexing of the CPU



- Transfer of control is called an activity-switch, i.e. depending on the type of activity:
  - Pure PULT switch (completely at user level)
  - Pure process switch
  - Pure KLT switch
  - Mixed switch, e.g. between a KLT and a process

Motivation

Why "thread states"?

Are these external thread states really necessary?

Do they at least enhance thread control?

If necessary, what thread states shall we implement?

First: Focus on KLT states

Later: PULT ~ and task states

Potential Benefits of KLT States

- Suppose you want to wake up a specific sleeping KLT
- You can find this KLT looking up the set of all KLTs
  - Assume t threads, i.e. O(t)
- If there is a subset containing only sleeping KLTs

⇒ You can wake up your sleepers in time

- In SMPs with a central ready queue and a global scheduling policy, KLT states are even necessary

⇒ Observation:

First place, we see a major difference between a single-processor and a multi-processor system

Enhanced TCB of a KLT in a SMP

Thread Identifier (TID)
Scheduling Thread State
Instruction Pointer (IP)
Stack Pointer (SP)
Status Flags (SF)

Either "Running"
or "Not Running"

Thread State Models

KLT Thread States

Remark:
The term thread state is a bit confusing because a running thread changes its “internal execution state” with every instruction

- This internal execution state of is called the KLTs context (see thread switching)

The term thread state represents the external relation of the KLT to its environment, i.e. to

- resources
- other KLTs
- …

KLT Thread State[1]

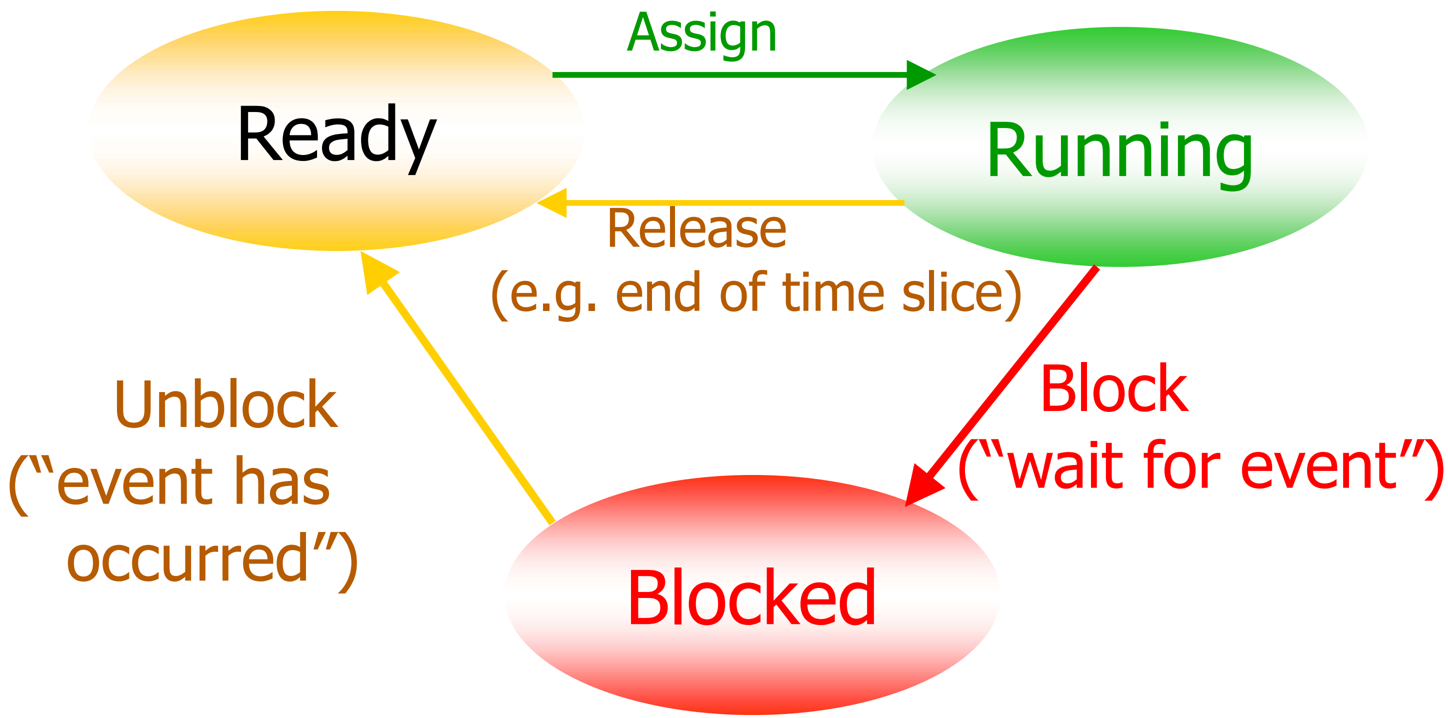
- A running thread is executing on a CPU
- A ready thread is a runnable thread, e.g. it could run, but it has no processor yet
- A blocked thread waits for an event to occur somewhere else, e.g.
  - end of previously initiated I/O
  - keyboard input
  - arrival of a message
  - arrival of a signal
  - release of a resource
  - ...

(end of previously initiated I/O, keyboard input: by a polled or interrupting device)

(arrival of a message, arrival of a signal, release of a resource: by another CPU activity)

[1]See process states

Three-State Thread Model

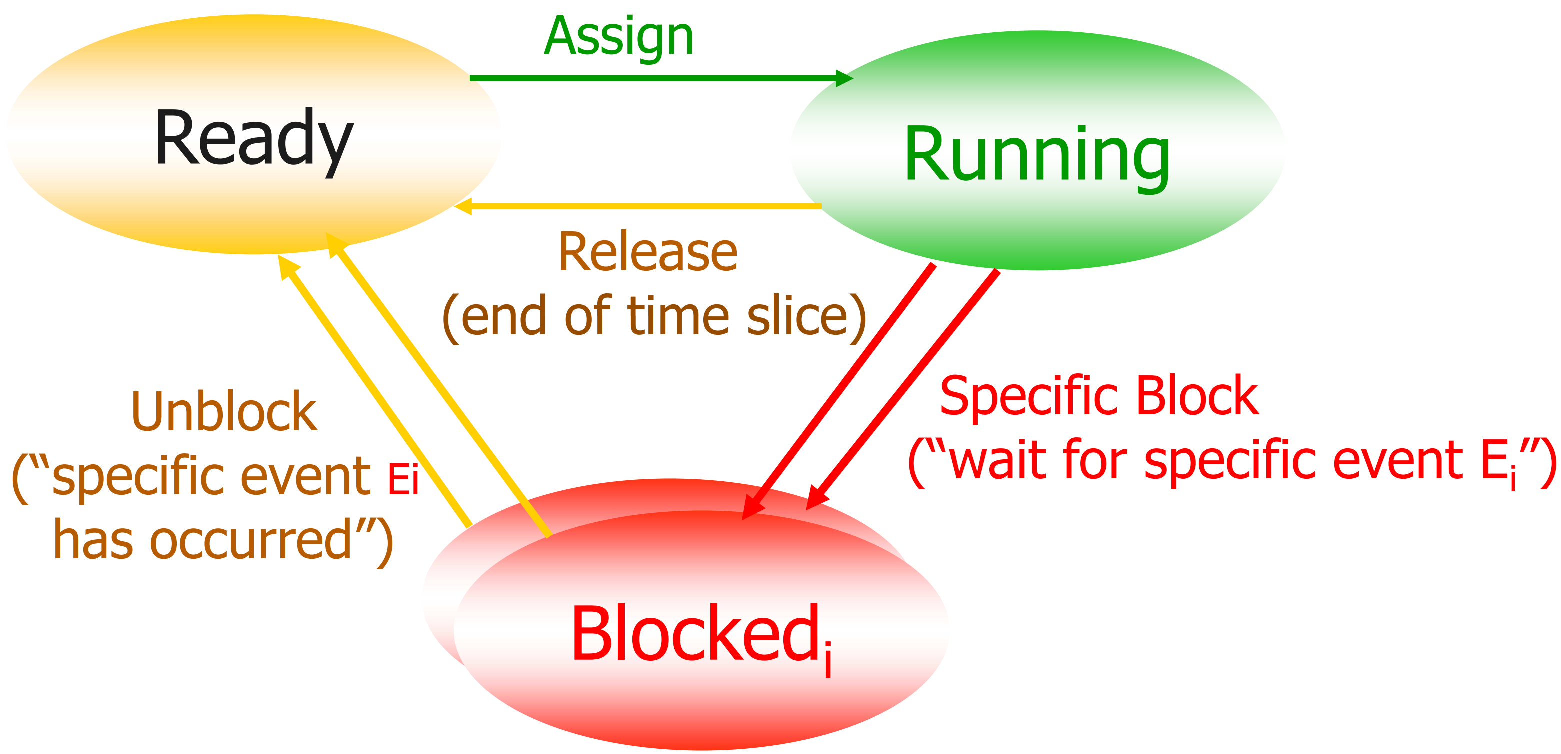


<u>Remark:</u>

Matter of design and not a matter of implementation whether ∃ only one state "blocked" for all waiting events

...

Three-State Thread Model



… or a separate KLT state blocked_i per event E_i

Additional KLT State

State "New"

- OS has created a KLT, i.e. it has
  - created a unique thread identifier
  - created a KLT TCB to manage the KLT
  - created corresponding AS entries (e.g. PTEs)
  - … created or initiated other needed system resources
- but OS has not yet committed to run the KLT (it is not yet admitted)* because
  - resources are limited or
  - ∃ some timing constraints, etc.

*Some claim that a modern OS needs an admission control

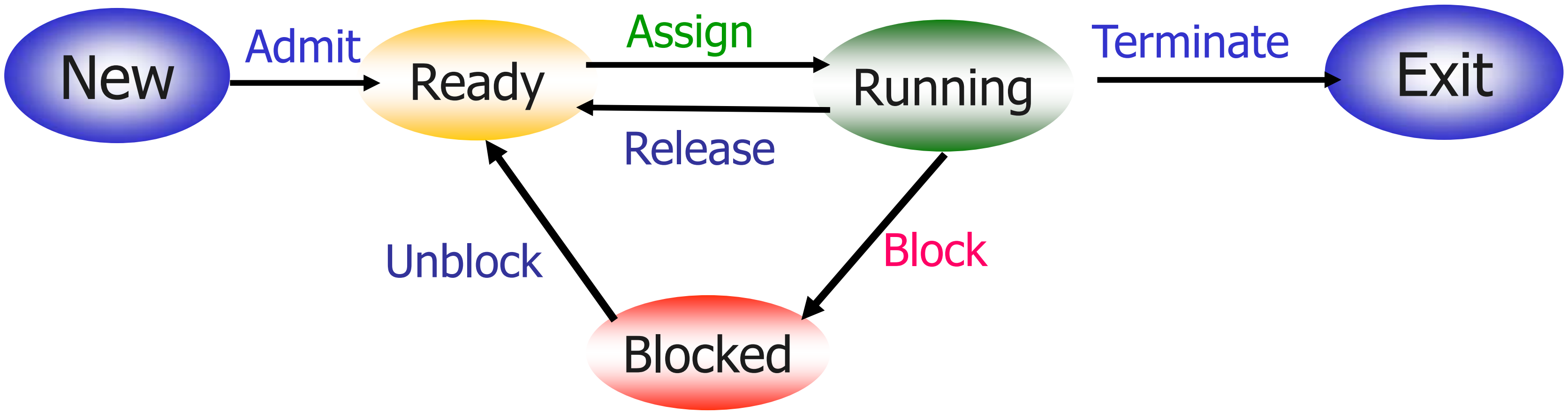
Additional KLT State

State "Exit"

- Thread no longer eligible for execution
- TCB, sub-tables and other info temporarily preserved for auxiliary programs
  - Example: accounting program that accumulates resource usage for billing its user
  - *When to delete code and stack and other thread specific regions in user space?*
- TCB (and its sub-tables) deleted when TCB entries are no longer needed

No answer to the question

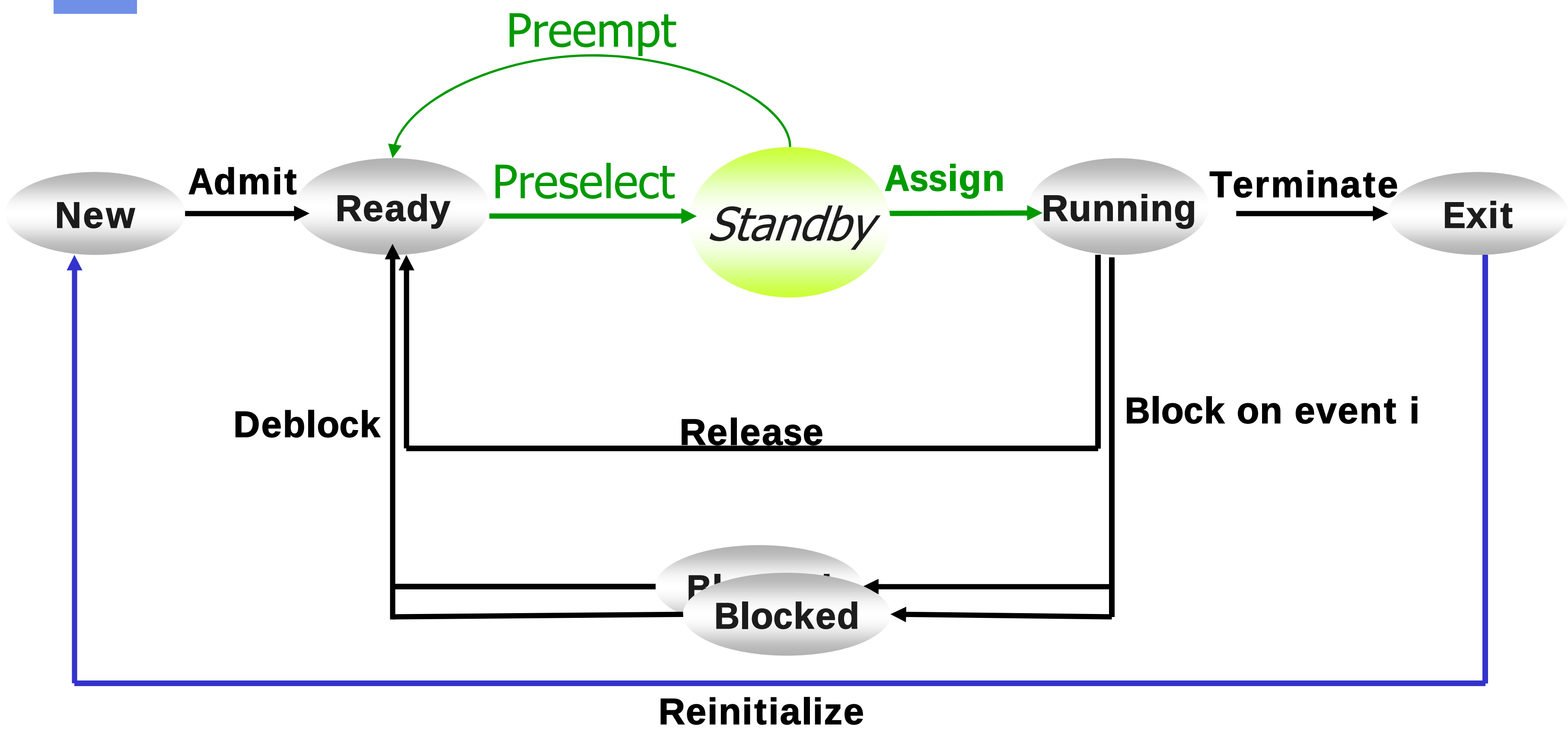
Five-State Thread Model



Remark:
∃ good reasons for introducing additional thread states, however, beware of overly complex "thread state models"

Design Rule 1: Keep Things Simple

Windows Six-State Thread Model



Why did MS system architects introduce KLT state standby?

- *Without it, can we do the job less or more elegantly?*
- *∃ other reasons for this thread state?*

Need for Swapping (States)

- In most systems complete tasks are mapped to RAM
- Even in a virtual memory system the following holds:
  - When too many applications are admitted at the same time, i.e. partially mapped to RAM, system performance decreases significantly (*thrashing phenomenon*)*
- If OS swaps out a complete KLT-task to disk, we have to distinguish:
  - Blocked Suspend: blocked threads that have been swapped out to disk or
  - Ready Suspend: ready threads that have been swapped out to disk

Implementing Thread State

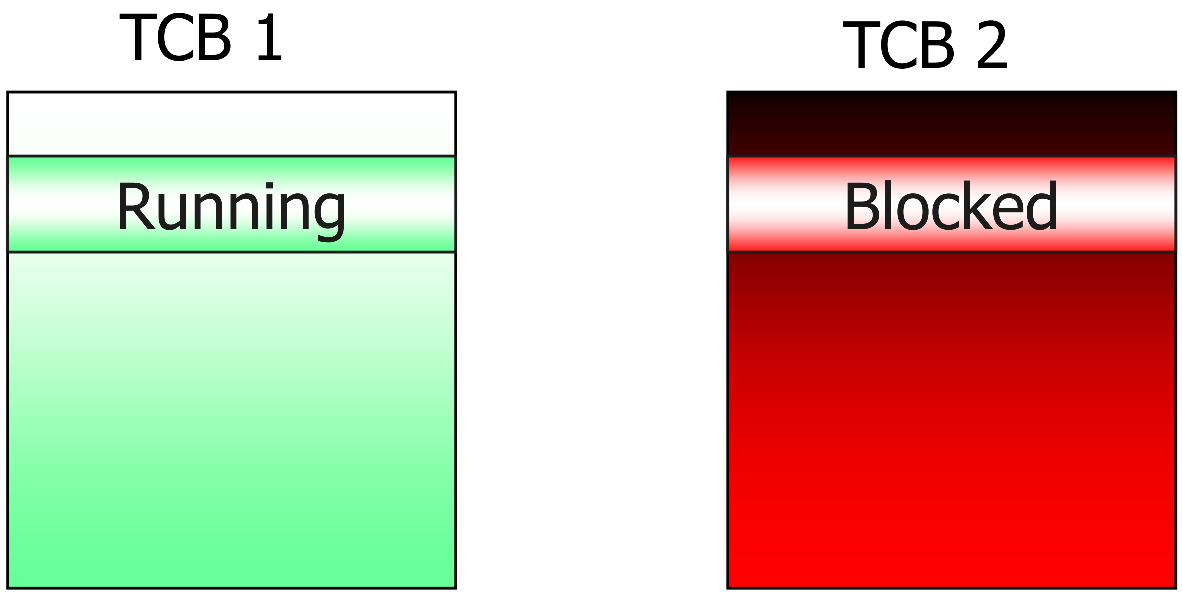
Implementing Thread States

- Another specific attribute (entry) in the TCB or
- An explicit data structure, e.g.
  - tree
  - double-linked list
  - Vector of dll
  - array …

Remark:
In some systems TCB attributes as well as explicit data structures are used to implement a specific thread state

Implementing Thread States

- Specific TCB attribute
- Explicit Data Structure

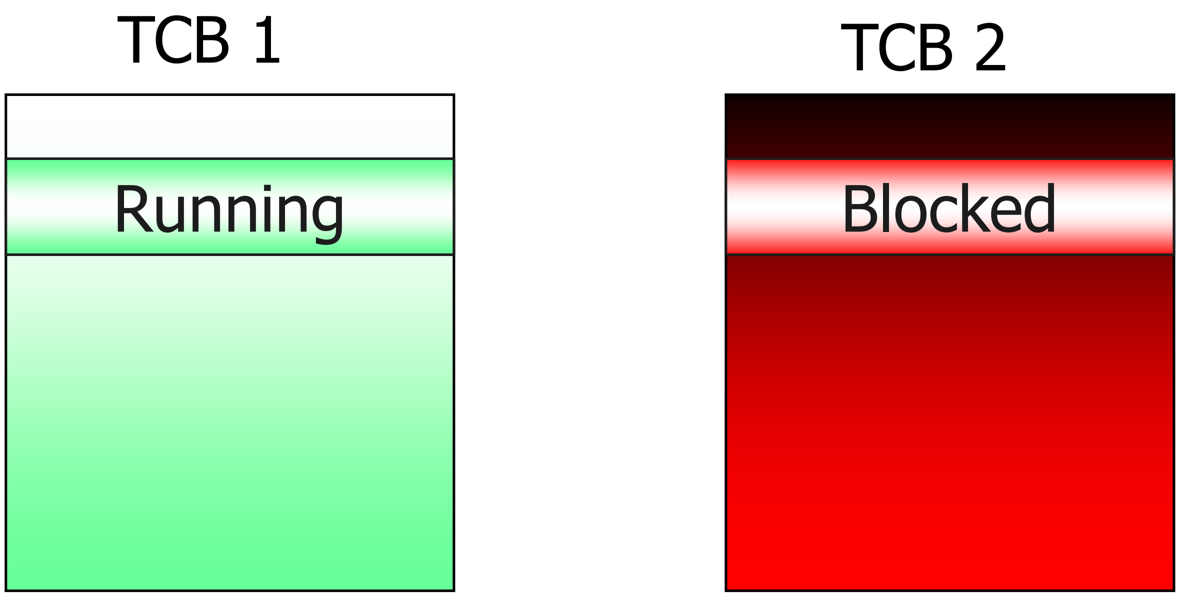


Discuss Pros and Cons

Thread State as a TCB Attribute

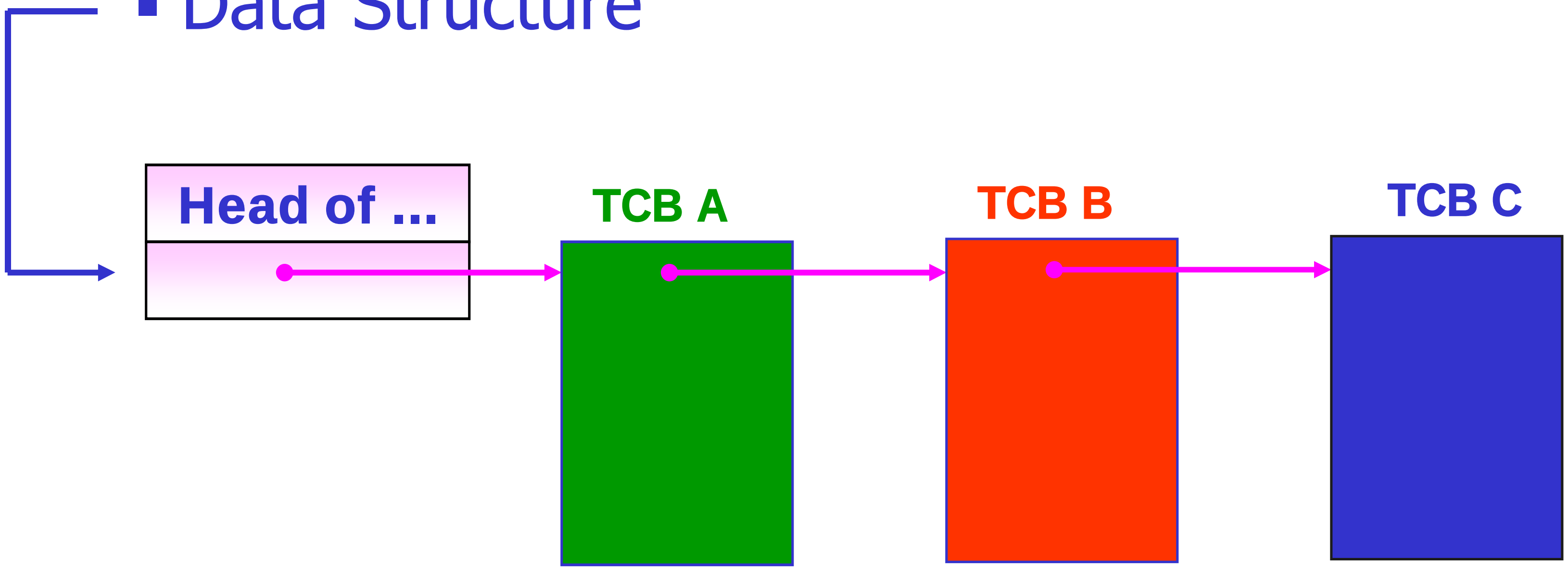
Obvious application:

1. Previous thread state for sake of *state history* or
2. An *intermediate thread state without* an extra subset implementation (see L4Ka)



Implementation of a Thread State

- Specific TCB attribute
- Data Structure



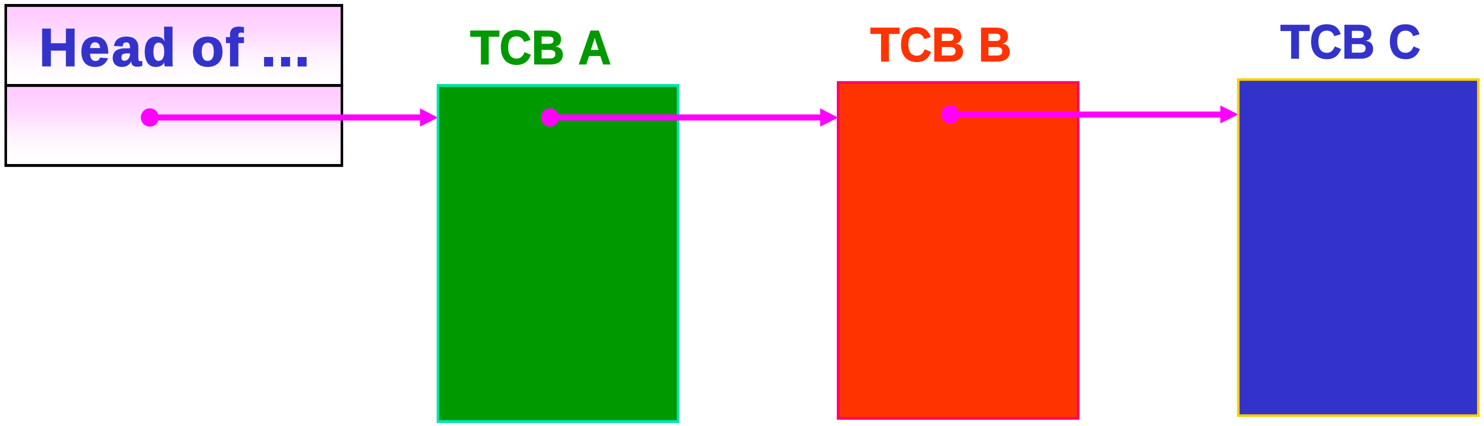
Discuss Pros and Cons!

Thread State via Data Structures

Obvious application:

Ready list = {threads which might be *running* next}

NT = first TCB after head of ready list (with O(1))



Rough Analysis 1

Assumption:

1. Given 1001 threads + 1 list for all threads
2. **No** attribute “thread state” within the TCB
3. **No** specific data structure for all *runnable* threads
4. Only CT is runnable, all other threads wait for events

Question: *Overhead for fair dispatching?*

A thread switch costs ~ 1 µsec

Result: 1000 thread switches in vain until previous running thread is dispatched again, i.e.

Overhead = 1000 µsec = “1 ms”

Rough Analysis 2

Assumption:

1. Given 1001 threads + 1 list for all threads
2. **Offer attribute "thread state" within the TCB**
3. **No** specific data structure for runnable threads
4. Only CT is runnable, all other threads wait for events

Question: Overhead for fair dispatching?

A thread switch still costs 1 µsec,
comparing 2 list entries ~ **0.1 µsec**

Result: 1000 additional comparisons in vain

Overhead = 101 µsec = "0.101 ms"

Rough Analysis 3

Assumption:

1. Given 1001 threads

2. Offer lists for runnable / *not runnable* threads

3. Only CT is runnable, all other threads wait for events

Question: *Overhead for fair dispatching?*

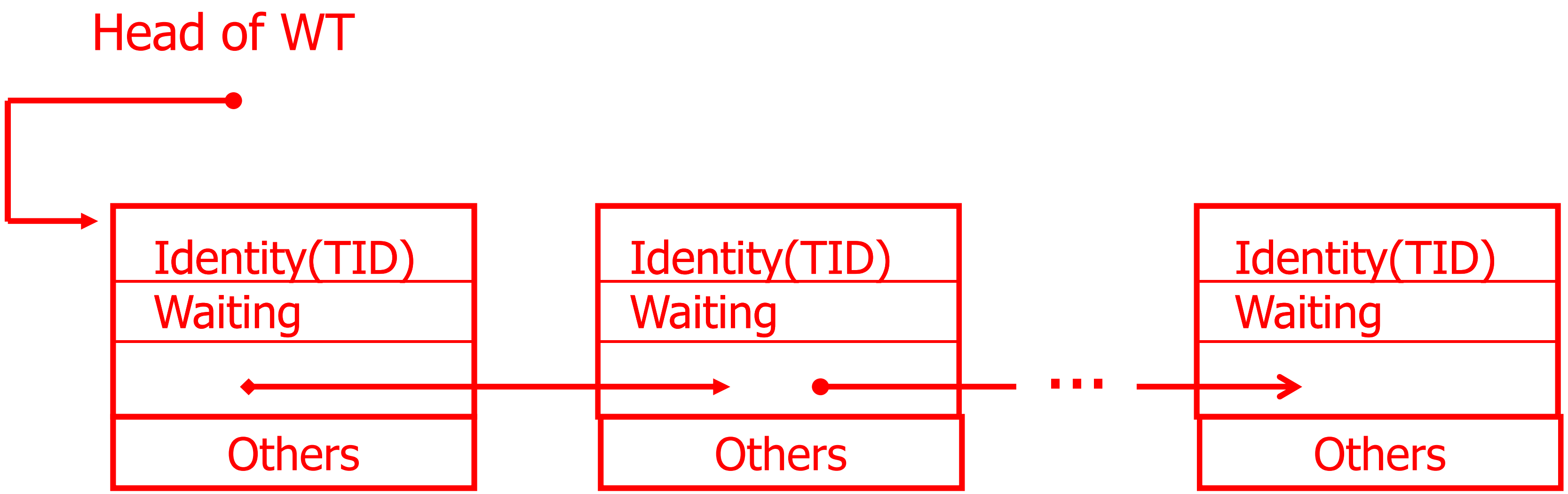
A thread switch costs 1 µsec,
comparing 2 list entries ~ 0.1 µsec

Result: Compare head of ready list, list empty
⇒ no thread switch is necessary

Overhead = "1.1 µsec"

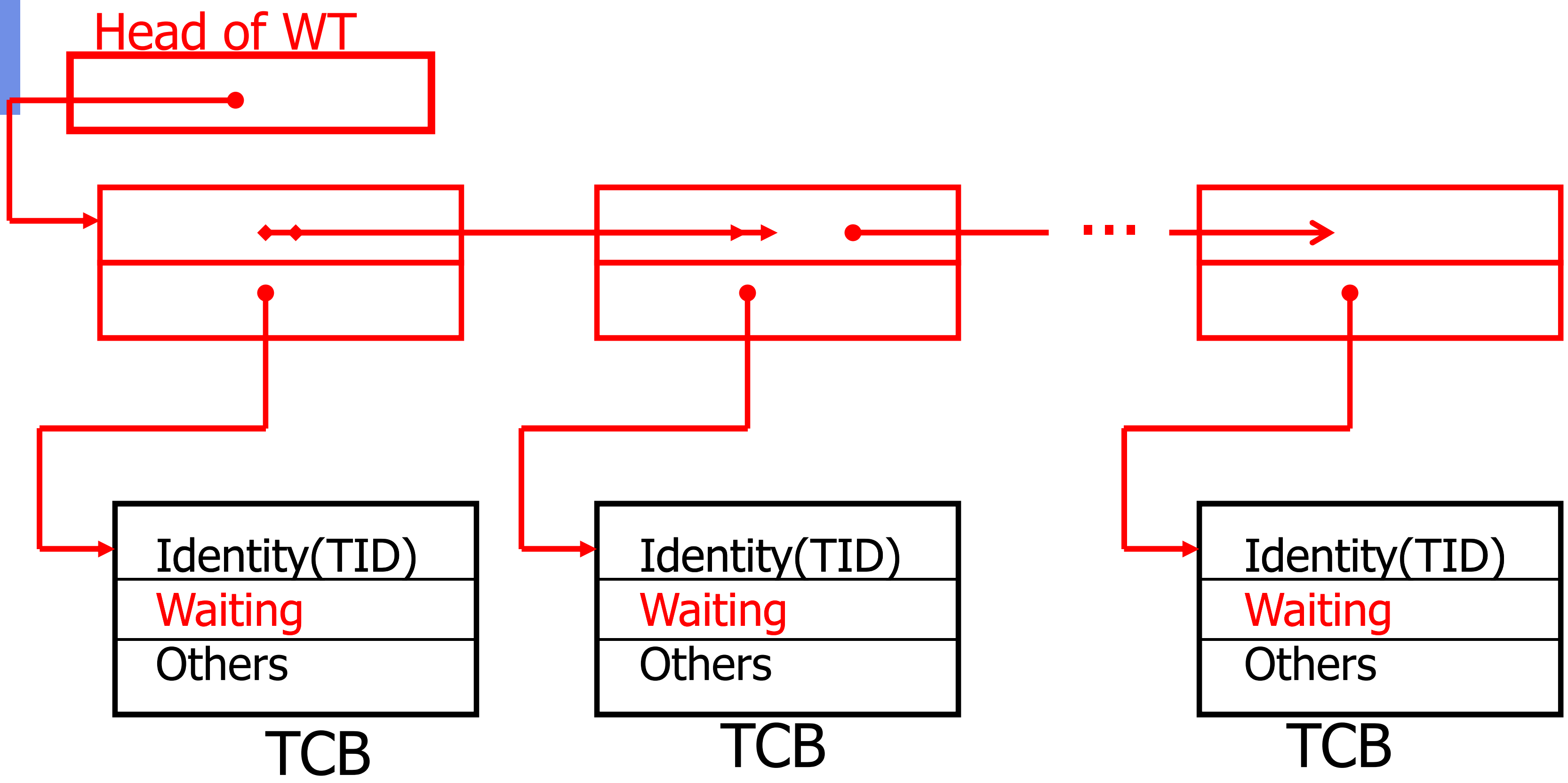
Pointers inside TCB

Waiting State = Some Type of a Queue*



*A single-linked list is often not a good choice at all

Pointers outside of TCB



Discuss Pros and Cons of this indirect method

Concluding Remarks

- If you chose a bad data structure for a frequently updated set of system entities, e.g. TCBs

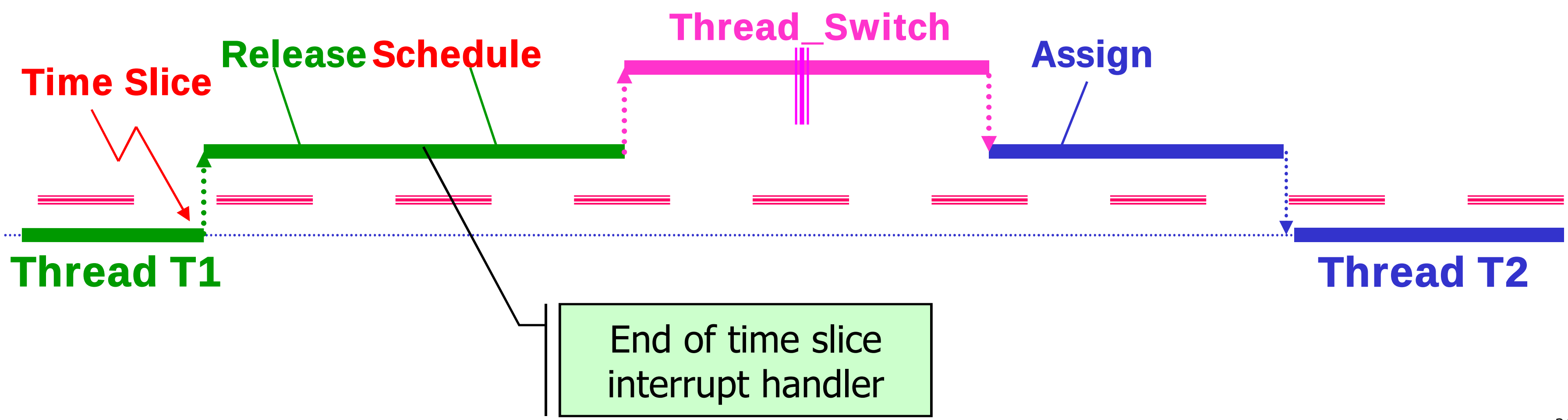
$\Rightarrow$ poor performance

- What is good for few threads (t < 16) can lead to a mission impossible for t >100, i.e. lack of scalability
- If we have to insert/delete at any position in the data set, a single linked list is one of the worst choices

Consequences for Dispatching

Consequences: 3-State Thread Model

```
interrupt procedure EoTS {End of Time Slice}
 begin
 …{time slice specific operations}
 Release(CT, SRT) {queue of ready threads}
 NT:=Schedule()                           {later}
 CT := ThreadSwitch(NT)
 Assign(CT)             {running thread(s)?}
 …{time slice specific operations}
 end
```



Implementing the Running Set

- On a single processor most processors have a specific register **`CURRENT`** pointing to the TCB of the running KLT (if not, you can define a specific pointer in the kernel AS to hold this address value)
- On a multi processor each processor has this register, **`CURRENT[i]`** but sometimes we need to know the load of the other processors as well
  - *When?*
- Implement an array of all relevant TCB attributes as the set of running KLTs

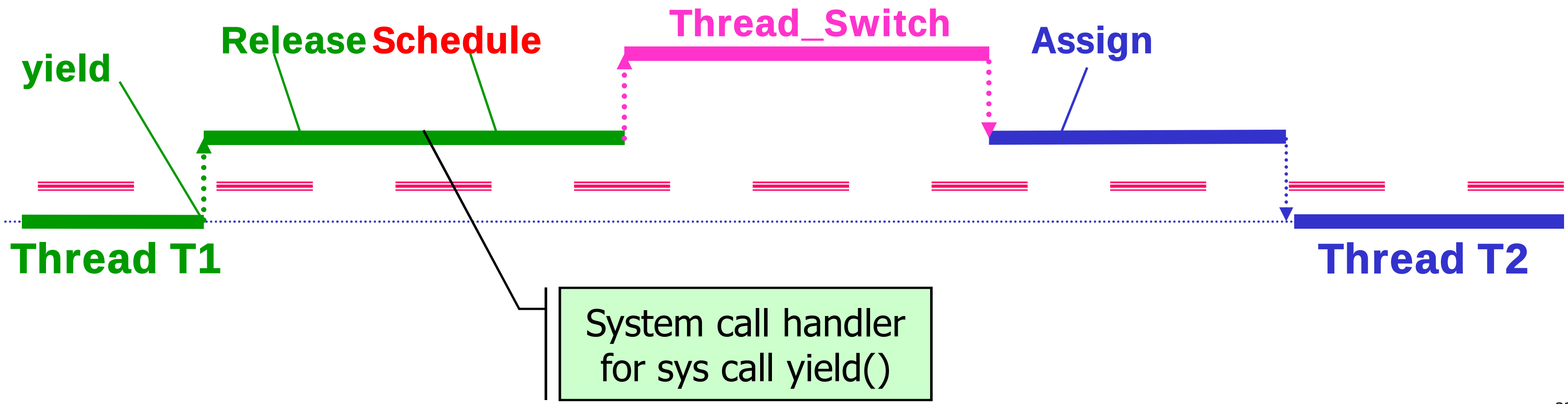
Consequences: 3-State Thread Model

Asynchronously & non voluntarily

```
interrupt procedure EoTS
 begin
 Release(CT,SRT)
 NT := Schedule()
 CT := ThreadSwitch(NT)
 Assign(CT)
 end
```

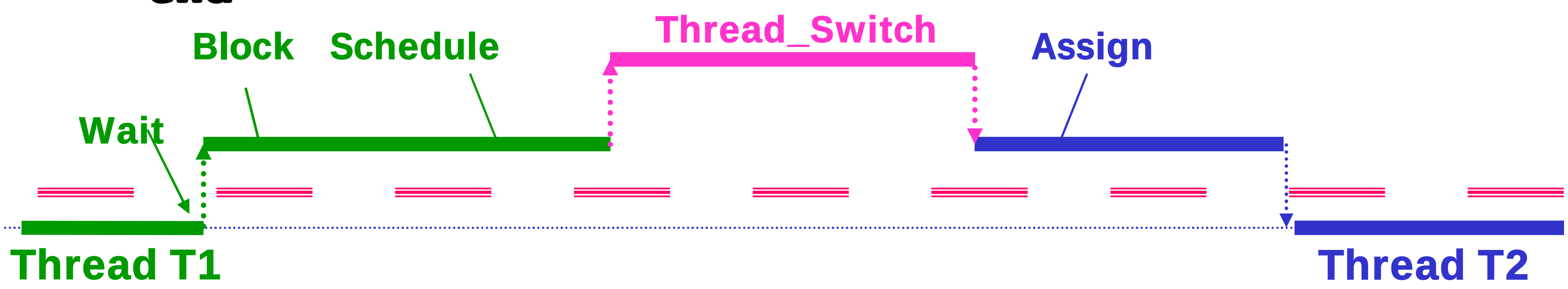
Synchronously & voluntarily

```
kernel procedure yield
 begin
 Release(CT,SRT)
 NT := Schedule()
 CT := ThreadSwitch(NT)
 Assign(CT)
 end
```



Consequences: 3-State Thread Model

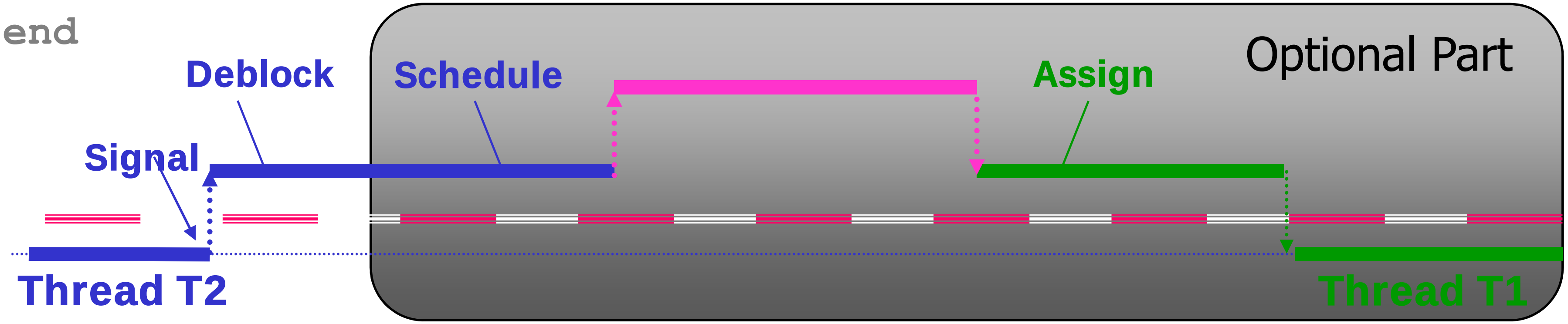
```
… procedure Wait(condition c)
  begin
  if c = true then     {if sometimes not sufficient}
  ...                  {remember case of just 1 state}
  Block(CT, c.SWT) {≠ BLOCK() see next chapter}
  NT := Schedule()
  CT := ThreadSwitch(NT)
  Assign(CT)
  else …
  fi
  end
```



Consequences: 3-State Thread Model

```
… procedure Wait(condition c)
begin
 if c = true then
  ...
  Block(CT,c.SWT)
   CT := Schedule(NT)
   ThreadSwitch(NT)
   Assign(CT)
  else
 ...
fi
end
```

```
… procedure Signal(condition c)
begin
 if c.SWT = non empty then
  ...
  Deblock(any(c.SWT),SRT)
  ...
 else
 ...
 fi
end
```

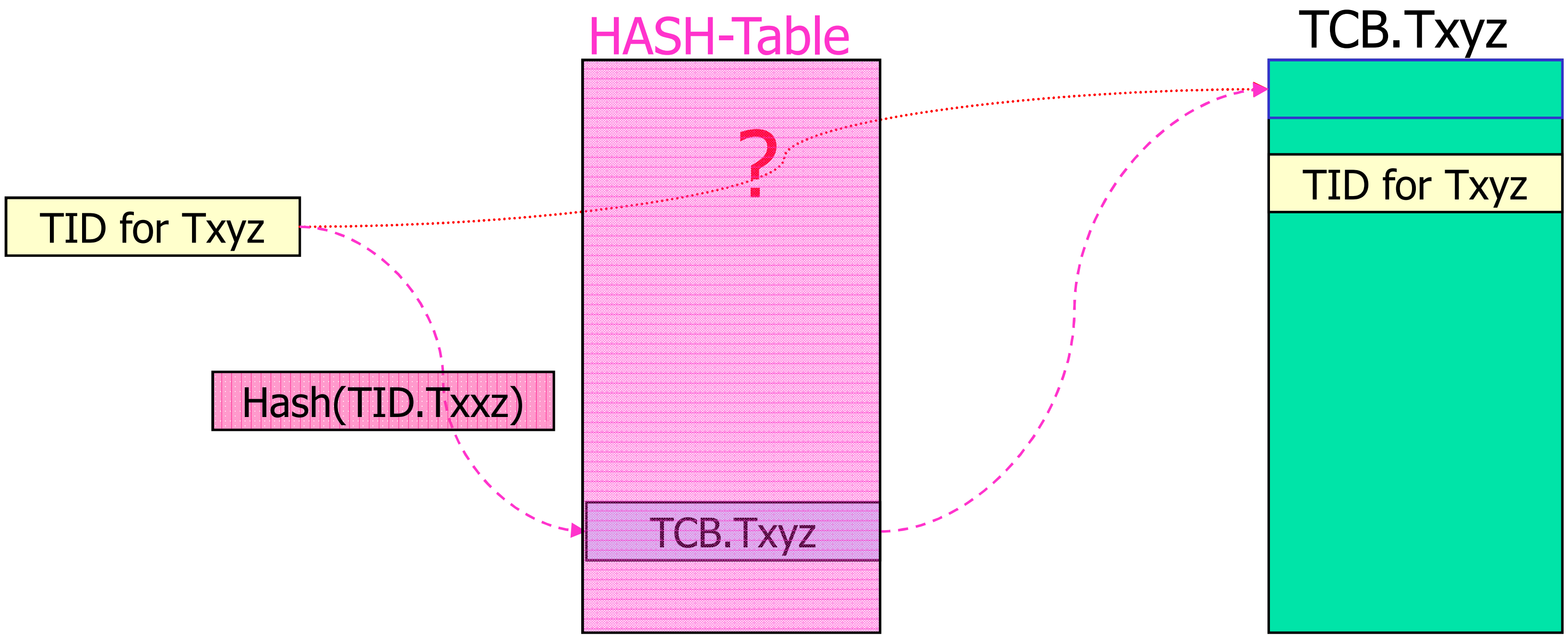


Preemption versus Non Preemption

- Without optional part scheduling policy is lazy
  - You do not deal with the fact that there is a new ready thread
  - There are system where you can do that
- There are systems that would result in a disaster if you would not react immediately whenever there is a change in the set of ready KLTs
  - Suppose a very urgent KLT has waited for a specific signal
  - Now this event happens, the signal handler unblocks this waiting KLT, i.e. it transfers the KLT from state "blocking" into the state "ready"
  - If you do not schedule, i.e. compare the urgency of the previously running KLT with the urgency of KLT you might risk life and limb

From TID to TCB?

- Some system calls need a TID as parameter, e.g. **`yield(NT)`** oder **`abort(child)`**
- *How to find the related TCB?*



Relation between Task & Thread States

Task States & KLT States

Task States & PULT States

Task & KLT States

Suppose a task T has *t>1 KLTs*, whereby *t-1* KLTs are currently *blocked* and only *1* KLT is either ready or *running*:

Is this task blocked or running or ready?

Related to the CPU the following holds:

running ≥ ready ≥ blocked, i.e.

KA specific

<u>Consequence:</u> As long as at least one KLT of a task is running ⇒ this task is running, regardless how many of its other KLTs are ready or even blocked

Kernel Activity for PULTs

Though kernel is not aware of a PULT, it manages its hosting task

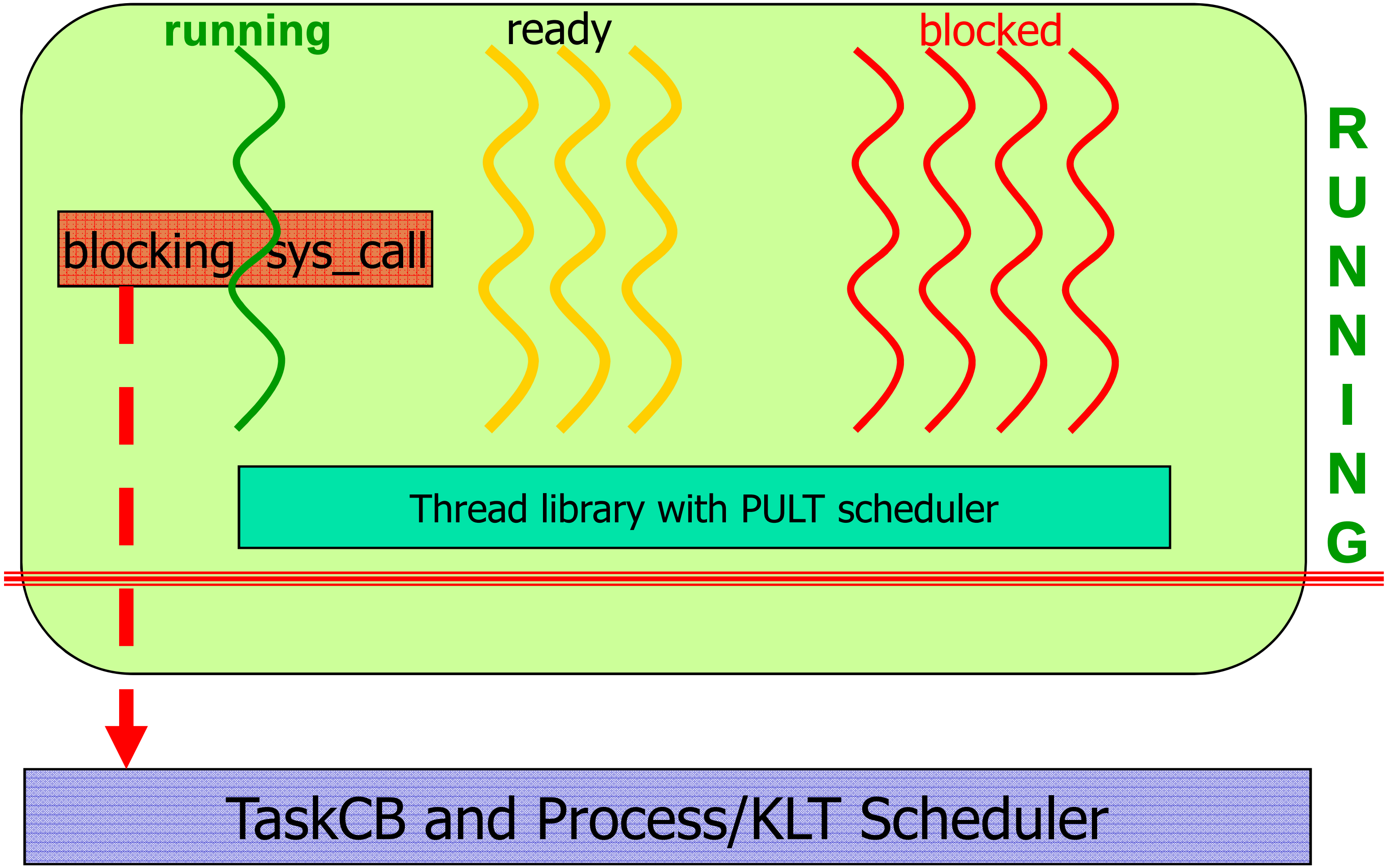
<u>Example:</u>

When a "PULT" does a "*blocking system call*" ⇒ the complete task will be blocked at kernel level

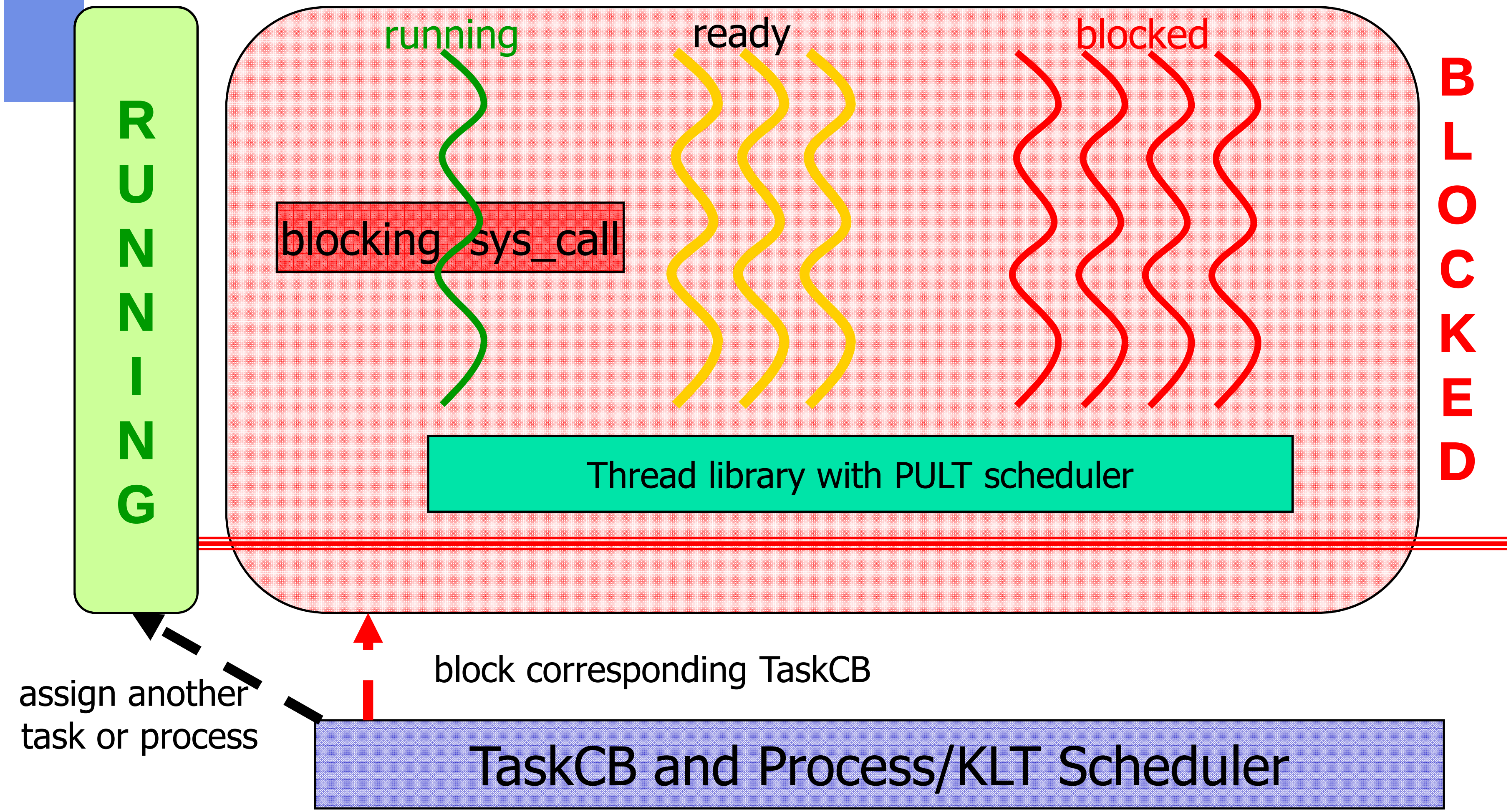
However, from the point of view of the user level scheduler that PULT is still "running" at user level

⇒ PULT states are independent of task states

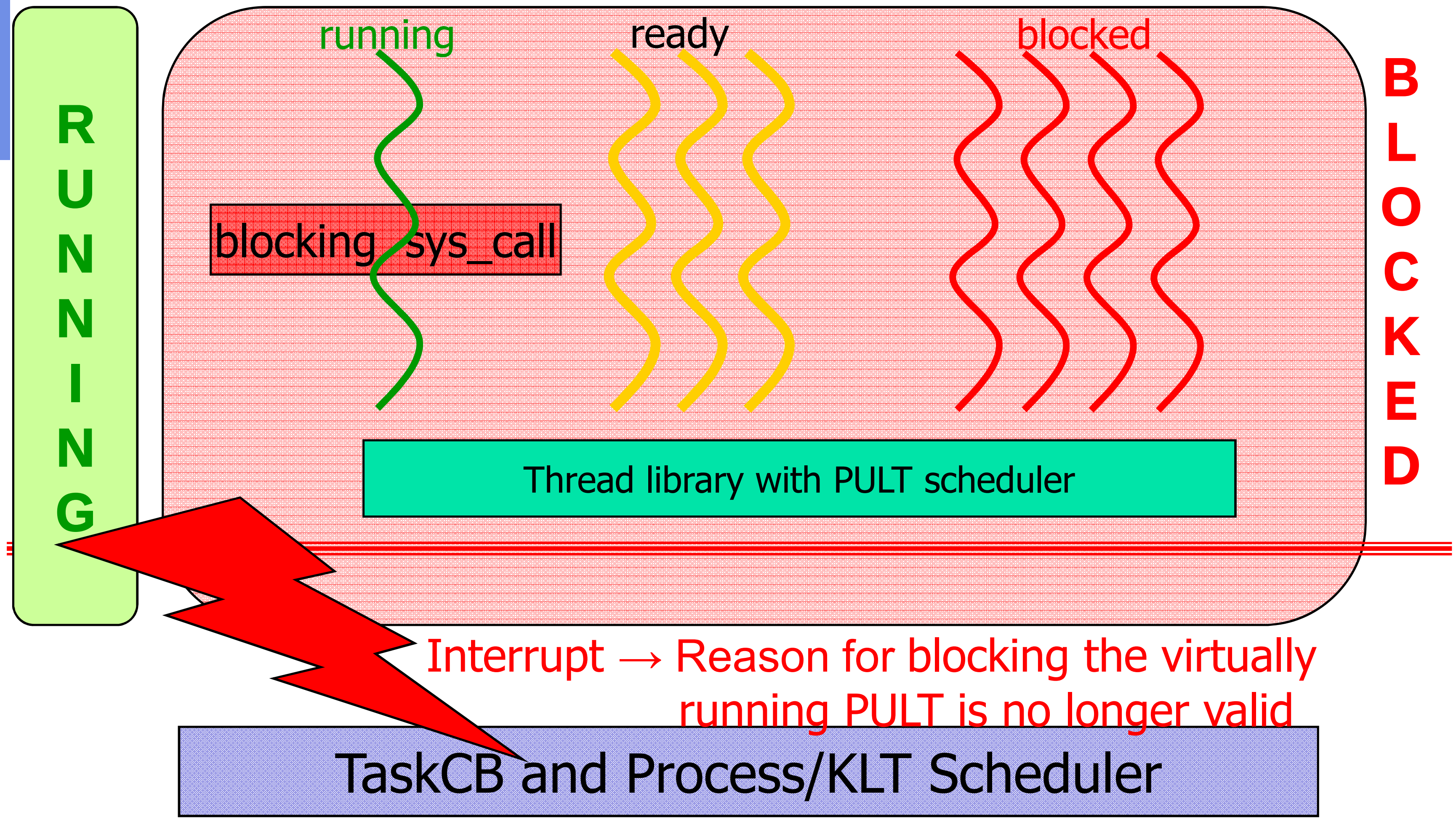
PULT & Task States



PULT- and Task-States

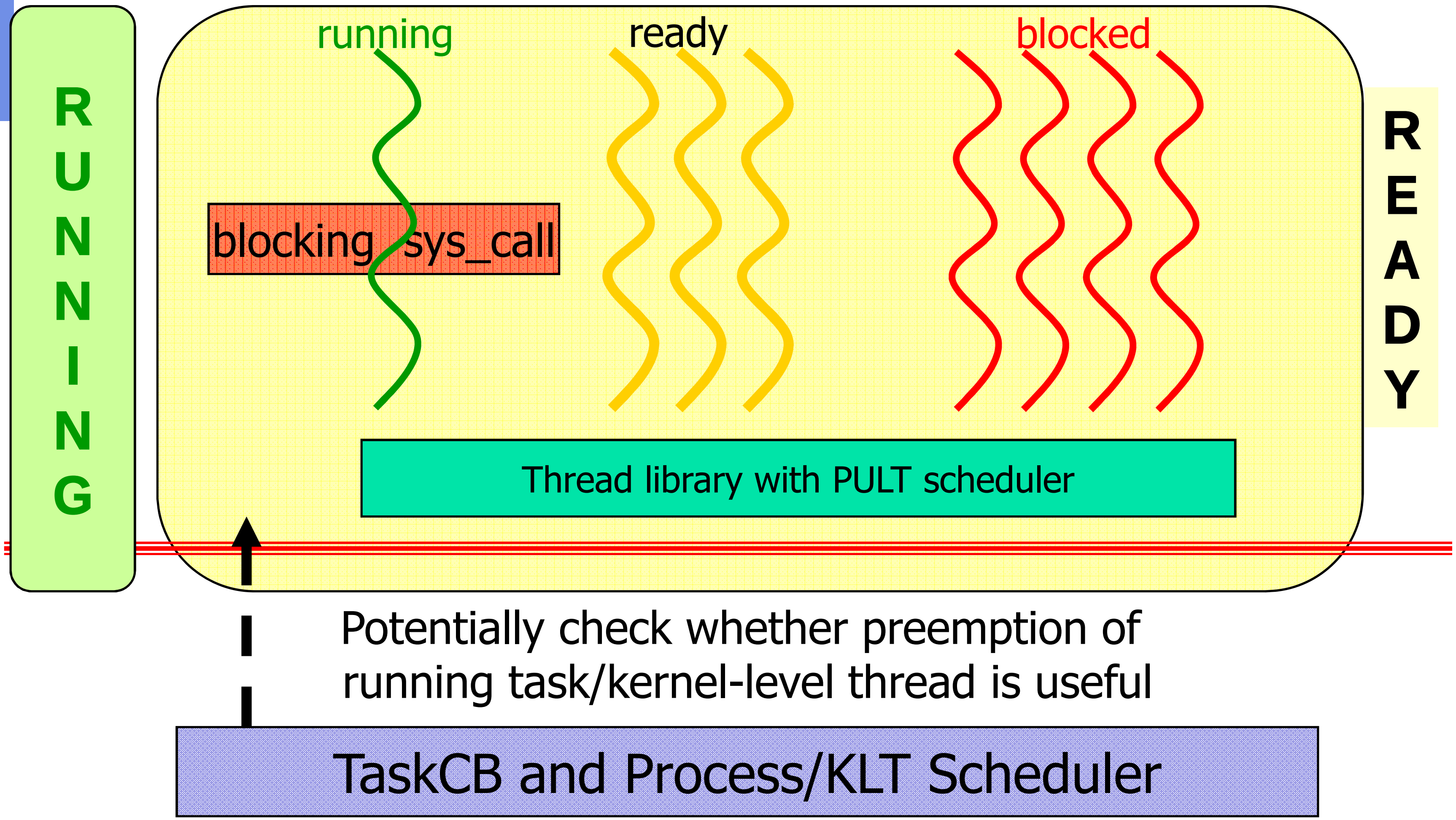


PULT- and Task-States



What happens next?

PULT- and Task-States



How can PULTs block at User-Level?

- ∃ thread library functions enabling a blocking (and unblocking) of a PULT at user-level, e.g.
  - In the Java-VM ∃ **`wait`** (and **`notify`**) to be used within a synchronized section (e.g. a method of a synchronized class)
  - Calling **`wait()`** blocks only the calling PULT and activates the library scheduler selecting the next ready PULT

What about Preemption?

How to prevent a PULT from hogging the CPU?

- Policy 1: No-Preemption
  - Requires cooperating PULTs
  - Each PULT must call back into the thread library periodically
    - Gives the library control over the threads' execution
  - `yield()` operation
    - The calling PULT voluntary gives up the CPU

What about Preemption?

How to prevent a PULT from hogging the CPU?

- Policy 2: Use Preemption
  - Thread library tells kernel to send a time signal periodically
    - Causes the task to jump into a signal handler
  - Signal handler gives control back to user level scheduler
    - User level scheduler selects next running thread and performs a PULT-switch

Summary

- Establish another thread state iff useful
- KLT-states & PULT-states ≠ task states (not always, but often)
- A PULT can be *running* (only virtually at user level) while its surrounding task is *blocked*
- A KLT can be *blocked* while other cooperating KLTs of the same task are *running, i.e.* while its task is still *running*

Cooperating Threads

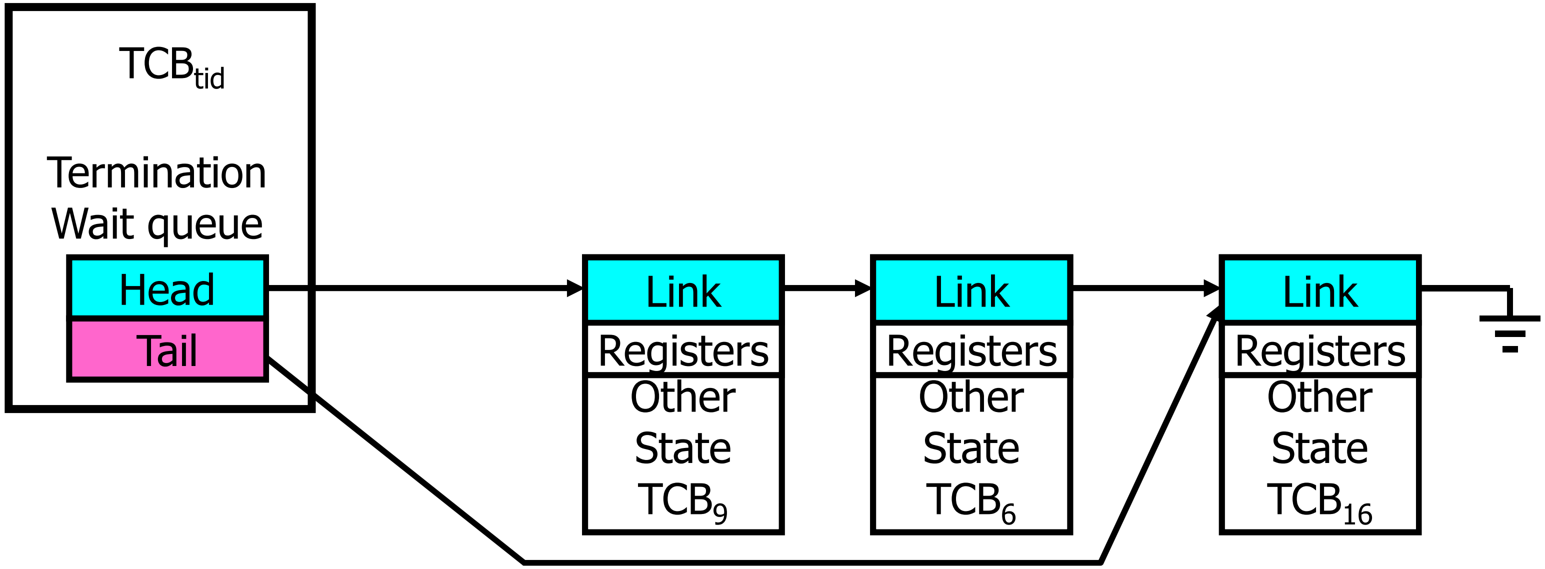
Forking

Thread Fork

- `ThreadFork(arg)` is not the same thing as UNIX `fork()`
  - UNIX `fork()` creates a new process (task) so it has to create a new address space
  - For now, don’t worry about how to create and switch between address spaces
- `Threadfork()` is ~ an asynchronous procedure call
  - Runs procedure `arg` in a separate thread in the same AS
  - Calling thread doesn’t wait for finish
  - If it want so it has to call it explicitly (e.g. `ThreadJoin`)
- *What if thread wants to exit early?*
  - `ThreadFinish()` and `exit()` are essentially the same procedure entered at user level

Thread Join

- One thread can wait for another to finish with the **`ThreadJoin(tid)`** call
  - Calling thread will be taken off the run queue and placed on waiting queue for thread **`tid`**
- Where is a logical place to store this wait queue?
  - On queue inside the TCB of **`tid`** ◄- ??



- Quite similar to **`wait()`** system call in UNIX
  - Lets parents wait for child processes

Use of Join for Procedures

- A traditional procedure call is logically equivalent to doing a **`ThreadFork()`** followed by **`ThreadJoin()`**
- Consider the following procedure call of **B()** by **A()**:

```
A() { B(); }
B() { Do interesting stuff }
```

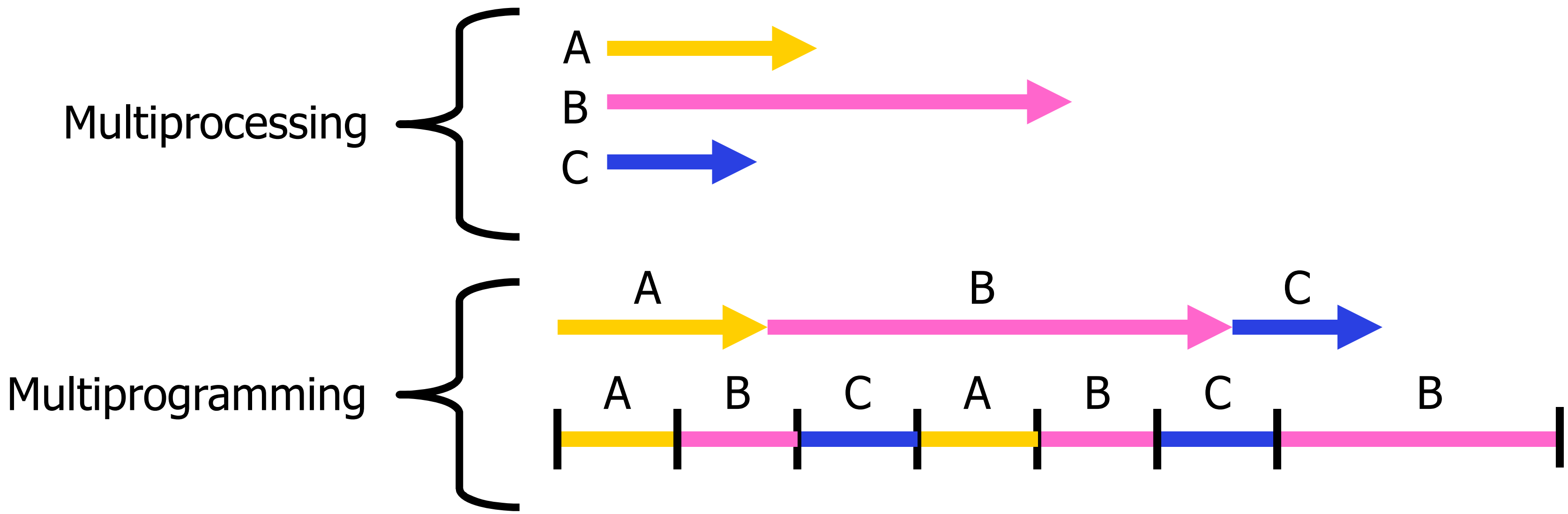
- The procedure **`A()`** is equivalent to **`A′():`**

```
A′() {
    tid = ThreadFork(B,null);
    ThreadJoin(tid);
}
```

- *Why not do this for every procedure?*
  - Context Switch Overhead
  - Memory Overhead for Stacks

Multi-Activity Models

- Multiprocessing ≡ Multiple CPUs
- Multiprogramming ≡ Multiple Jobs or Processes
- Multithreading ≡ Multiple threads per Task
- *What does it mean to run two threads "concurrently"?*
  - Scheduler is free to run threads in any order and interleaving: FIFO, Random, …
  - Dispatcher can choose to run each thread to



Correctness with Threads

- If a dispatcher can schedule threads in any way, programs must work under all circumstances
  - *Can you test for this?*
  - *How can you know if your program works?*

- Independent Threads:
  - No state shared with other threads
  - Deterministic ⇒ input state determines results
  - Reproducible ⇒ can recreate initial conditions, I/O
  - Scheduling order doesn't matter (if **`switch()`** works!!!)

Correctness with Threads

- Cooperating Threads:
  - Shared State between multiple threads
  - Non-deterministic
  - Non-reproducible
- Non-deterministic and non-reproducible means that bugs can be intermittent
  - Sometimes called "***Heisenbugs***"

Interactions & Debugging

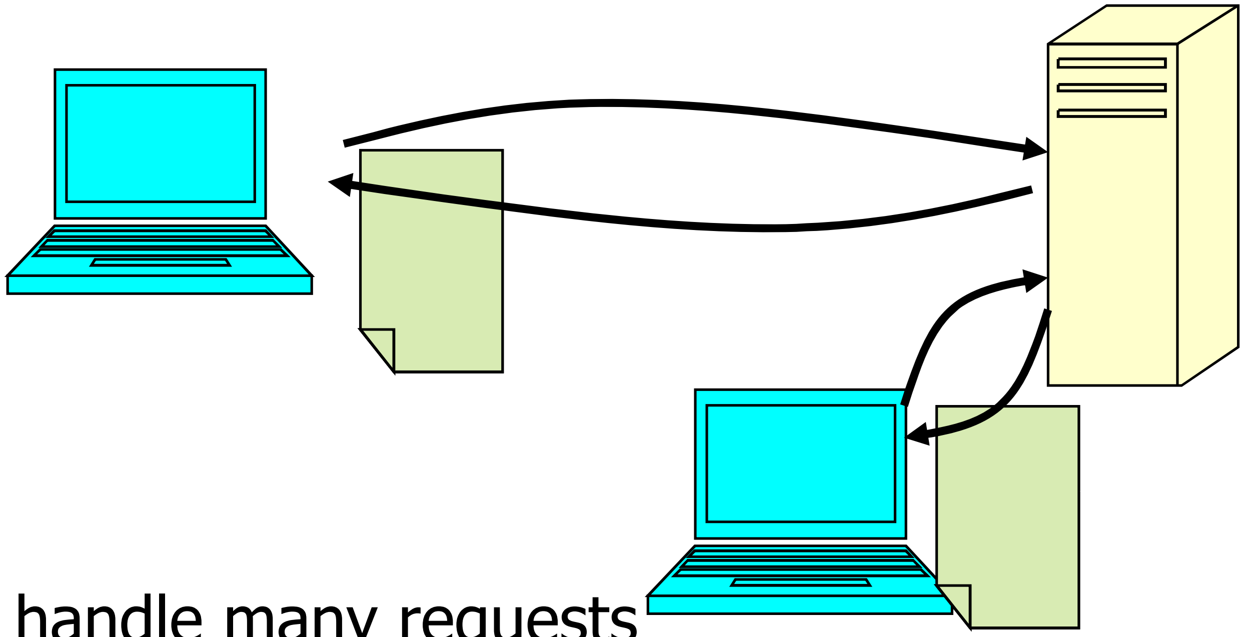
- Is any program truly independent?
  - Every process shares the file system, OS resources, network, etc.
  - Extreme example: buggy device driver causes thread A to crash "independent thread" B
- You probably don't realize how much you depend on reproducibility:
  - Example: Evil C compiler
    - Modifies files behind your back by inserting errors into C program unless you insert debugging code
  - Example: Debugging statements can overrun stack
- Non-deterministic errors are really difficult to find
  - Example: Memory layout of kernel + user programs
    - depends on scheduling, which depends on timer/other things
    - Original UNIX had a bunch of non-deterministic errors

Why Cooperating Threads?

People cooperate; computers help/enhance people’s lives, that’s why computers must cooperate

- Advantage 1: Share resources
  - One computer, many users
  - One bank balance, many ATMs
    - What if ATMs were only updated at night?
  - Embedded systems (robot control: coordinate arm & hand)
- Advantage 2: Speedup
  - Overlap I/O and computation
    - Many different file systems do read-ahead
  - Multiprocessors – chop up program into parallel pieces
- Advantage 3: Modularity
  - More important than you might think
  - Chop a large problem up into simpler pieces
    - To compile, for instance, gcc calls cpp | cc1 | cc2 | as | ld
    - Makes system easier to extend

Example: Web Server



- Server must handle many requests
- Non-cooperating version:

```
serverLoop() {
    con = AcceptCon();
    "ProcessFork"(ServiceWebPage(),con);
}
```

- *What are some disadvantages of this technique?*

Multi-Threaded Web Server

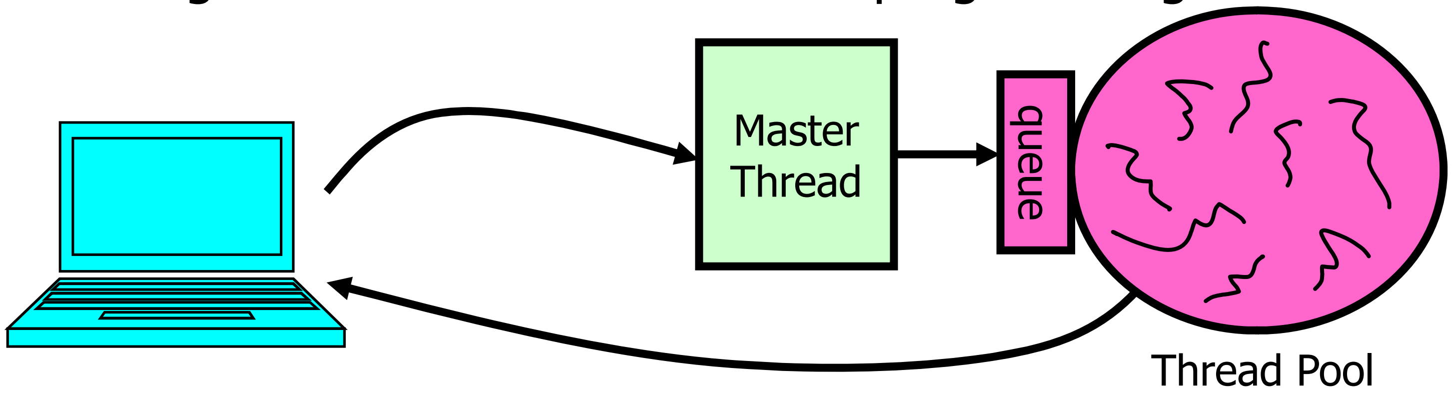
- Now, use a single process
- Multithreaded (cooperating) version:

```
serverLoop() {
      connection = AcceptCon();
      ThreadFork(ServiceWebPage(),connection);
  }
```

- Looks almost the same, but has many advantages:
  - Can share file caches kept in memory, results of CGI scripts, other things
  - Threads are *much* cheaper to create than processes, so this has a lower per-request overhead
- Question: would a user-level (say one-to-many) thread package make sense here?
  - When one request blocks on disk, all block…
- *What about Denial of Service attacks or digg / Slash-dot effects?*

(Un)Limited Thread Pools

- Problem with previous version: Unbounded Threads
  - When web-site becomes too popular – throughput slows down
- Instead, allocate a bounded "pool" of worker threads, representing the maximum level of multiprogramming




```
master() {
   allocThreads(worker,queue);
   while(TRUE) {
      con=AcceptCon();
      Enqueue(queue,con);
      wakeUp(queue);
   }
}
```

```
worker(queue) {
   while(TRUE) {
      con=Dequeue(queue);
      if (con==null)
         sleepOn(queue);
      else
         ServiceWebPage(con);
   }
}
```

Summary

- Interrupts = HW mechanism for returning control to OS kernel
  - Used for important/high-priority peripheral events
  - Can force dispatcher to schedule a different thread (preemptive multithreading)
- New Threads Created with `ThreadFork()`
  - Create initial TCB and stack to point at `ThreadRoot()`
  - `ThreadRoot()` calls thread code, then `ThreadFinish()`
  - `ThreadFinish()` wakes up waiting threads then prepares TCB/stack for destruction
- Threads can wait for other threads using `ThreadJoin()`
- Threads may be "implemented" as user-level or kernel level
- Cooperating threads have many potential advantages
  - But: introduces non-reproducibility and non-determinism
  - Need to have **atomic operations**

Recommended Reading

- Bacon, J.: Operating Systems (4)
- Nehmer, J.: Grundlagen moderner BS (5.2)
- Silberschatz, A.: Operating System Concepts (2)
- Stallings, W.: Operating Systems (3, 4)
- Tanenbaum, A.: Modern Operating Systems (2)
- Vogt, C.: Betriebssysteme (3)