University of Glasgow | School of Computing Science

Implementing Real-time Systems

Advanced Operating Systems (M)
Lecture 9

Lecture Outline

- Implementing real time systems
  - Key concepts and constraints
  - System architectures:
    - Cyclic executive
    - Microkernel with priority scheduler
- Implementing scheduling algorithms
  - Jobs, tasks, and threads
  - Priority scheduling of periodic tasks
    - Rate monotonic
    - Earliest deadline first
  - Priority scheduling of aperiodic and sporadic tasks

Implementing Real Time Systems

- Key fact from scheduler theory: need predictable behaviour
  - Raw performance less critical than consistent and predictable performance; hence focus on scheduling algorithms, scheduling tests
  - Don't want to fairly share resources – be unfair to ensure deadlines met
- Need to run on a wide range of – often custom – hardware
  - Often resource constrained:
    - limited memory, CPU, power consumption, size, weight, budget
  - Embedded and may be difficult to upgrade
    - Closed set of applications, trusted code
    - Strong reliability requirements – may be safety critical
    - How to upgrade software in a car engine? A DVD player? After you shipped millions of devices?

Implications on Operating Systems

- General purpose operating systems not well suited for real time
  - Assume plentiful resources, fairly shared amongst untrusted users
  - Exactly the opposite of an RTOS!
- Instead want an operating system that is:
  - Small and light on resources
  - Predictable
  - Customisable, modular and extensible
  - Reliable
  - …and that can be demonstrated or proven to be so

Implications on Operating Systems

- RTOS often use cyclic executive or microkernel designs, rather than a traditional monolithic kernel
  - Limited and well defined functionality
  - Easier to demonstrate correctness
  - Easier to customise
- Provide rich scheduling primitives
- Provide rich support for concurrency
- Expose low-level system details to the applications
  - Control of scheduling
  - Power awareness
  - Interaction with hardware devices

Cyclic Executive

- The simplest RTOS use a “nanokernel” design
  - Provides a minimal time service: scheduled clock pulse with fixed period
  - No tasking, virtual memory/memory protection, etc.
  - Allows implementation of a static cyclic schedule, provided:
    - Tasks can be scheduled in a frame-based manner
    - All interactions with hardware to be done on a polled basis
  - OS becomes a single task cyclic executive

```
setup timer
c = 0;
while (1) {
    suspend until timer expires
    c++;
    do tasks due every cycle
    if (((c+0) % 2) == 0)
       do tasks due every 2nd cycle
    if (((c+1) % 3) == 0)
       do tasks due every 3rd cycle, with phase 1
    ...
}
```

Microkernel Architecture

- Cyclic executive widely used in low-end devices
  - 8 bit processors with kilobytes of memory, programmed in C via cross-compiler, simple hardware interactions, simple static task set
- But, many real-time systems are more complex, and need a sophisticated operating system with priority scheduling
  - Common approach: a microkernel with priority scheduler
  - Configurable and robust, since architected around interactions between cooperating system servers, rather than a monolithic kernel with ad-hoc interactions

Microkernel Architecture

- A microkernel RTOS typically provides a number of features:
  - Scheduling
  - Timing services, interrupt handling, support for hardware interaction
  - System calls with predictable timing behaviour
  - Messaging, signals and events
  - Synchronisation and locking
  - Memory protection

- Details often differ from non-RTOS environments

Scheduler Implementation

- Clock driven scheduling trivial to implement via cyclic executive

- Other scheduling algorithms need operating system support:
  - System calls to create, destroy, suspend and resume tasks
  - Implement tasks as either threads or processes
  - Processes (with separate address space and memory protection) not always supported by the hardware, and *often not useful*
  - Scheduler with multiple priority levels, range of periodic task scheduling algorithms, support for aperiodic tasks, support for sporadic tasks with acceptance tests, etc.

Jobs, Tasks and Threads

- A system comprises a set of *tasks*, each task is a series of *jobs*
  - Tasks are typed, have various parameters (φ, p, e, D), react to events, etc.
  - Acceptance test performed before admitting new tasks
- A thread is the basic unit of work handled by the scheduler
  - Threads are the instantiation of tasks that have been admitted to the system
- How are tasks and jobs mapped onto threads and managed by the scheduler?

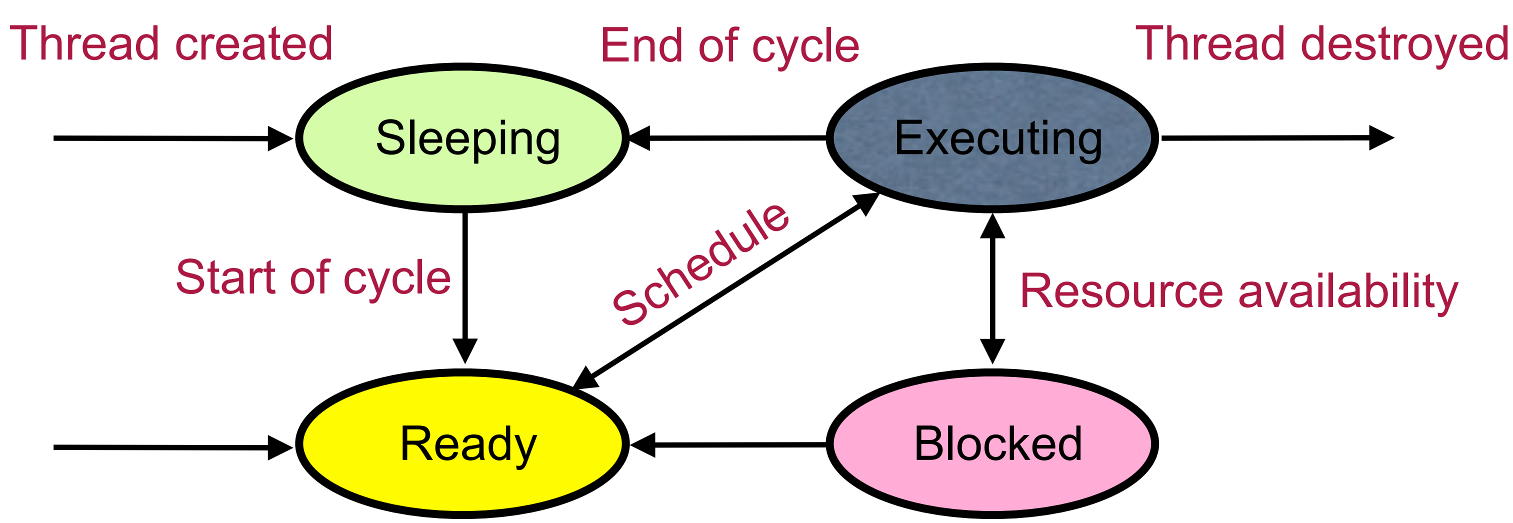
Periodic Tasks

- Real time tasks defined to execute periodically
  - $T = (\varphi, p, e, D)$
- Two implementation strategies:
  - Thread instantiated by system each period, runs single job
    - A periodic thread ⇒ supported by some RTOS
    - Clean abstraction: a function that runs periodically; system handles timing
    - High overhead due to repeated thread instantiation, although thread pools can mitigate overhead
  - Thread instantiated once, performs job, sleeps until next period, repeats
    - Lower overhead, but relies on programmer to handle timing
    - Pushes conceptual burden of handling timing onto programmer
    - Hard to avoid timing drift due to sleep overruns
    - Most common approach

Sporadic and Aperiodic Tasks

- Event list triggers sporadic and aperiodic tasks
  - Might be external (hardware) interrupts
  - Might be signalled by another task
- Several implementation strategies:
  - Job runs as interrupt/signal handler
    - Can be disruptive for other real-time tasks
    - Handler often used to instantiate sporadic thread or queue job for server task
  - Thread instantiated by system when job released
    - Not well supported for user-level jobs, often used within the kernel (e.g., for device drivers; network processing)
    - Requires scheduler assistance; high overheads unless thread pool used
  - Job queued for server task
    - A background server (simple, widely implemented)
    - A bandwidth preserving server (useful, but hard to implement)

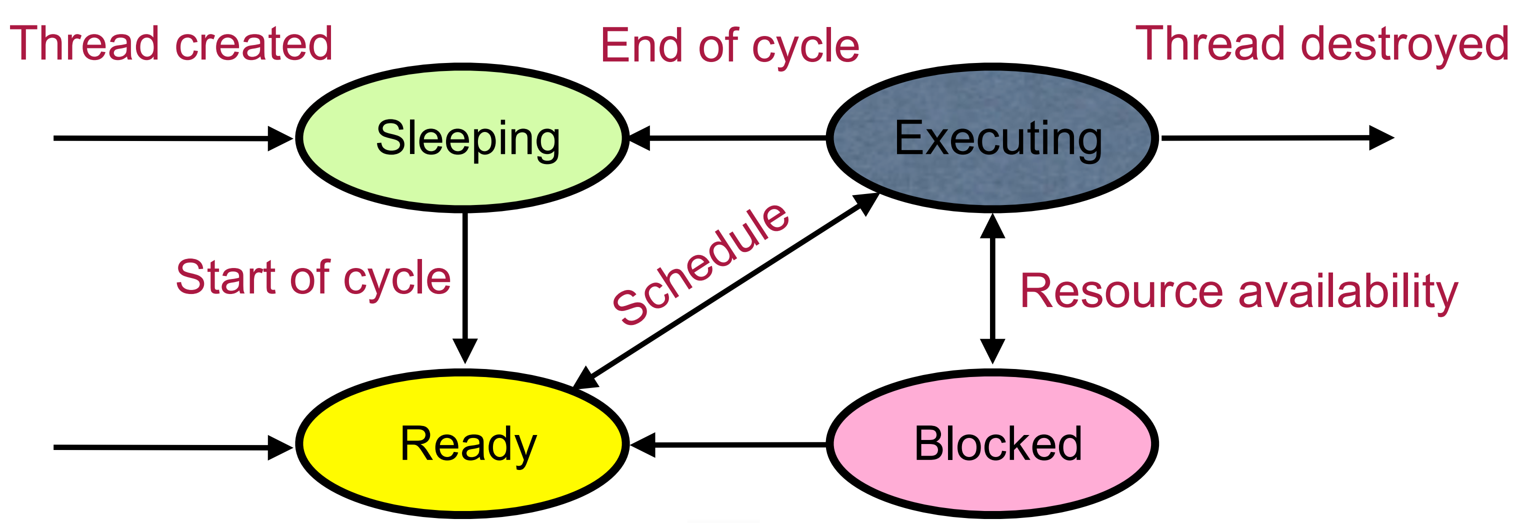
Thread States and Transitions

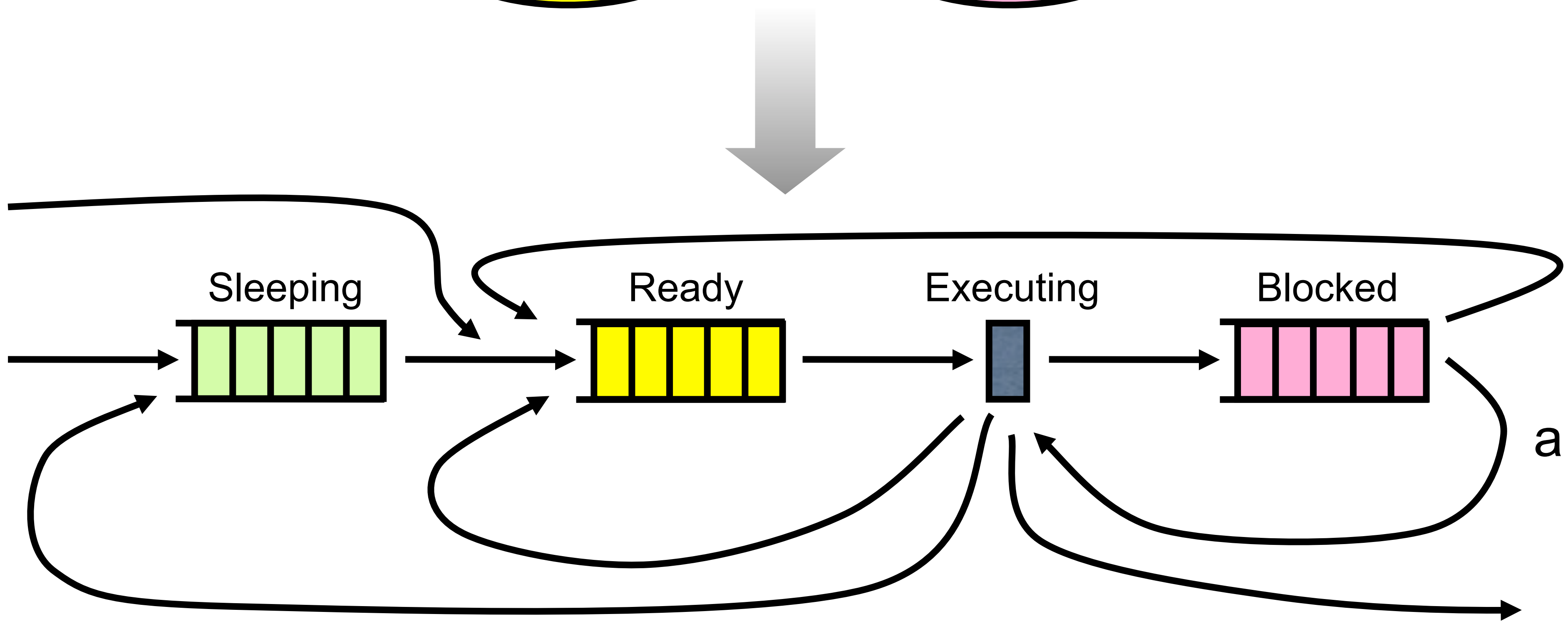


- States represent evolution of thread execution:
  - Sleeping ⇒ Periodic thread queued between cycles
  - Ready ⇒ Queued at some priority, waiting to run
  - Executing ⇒ Running on a processor
  - Blocked ⇒ Queued waiting for a resource
- Transitions happen according to scheduling policy, resource access, external events

Mapping States onto Queues

Abstract states…

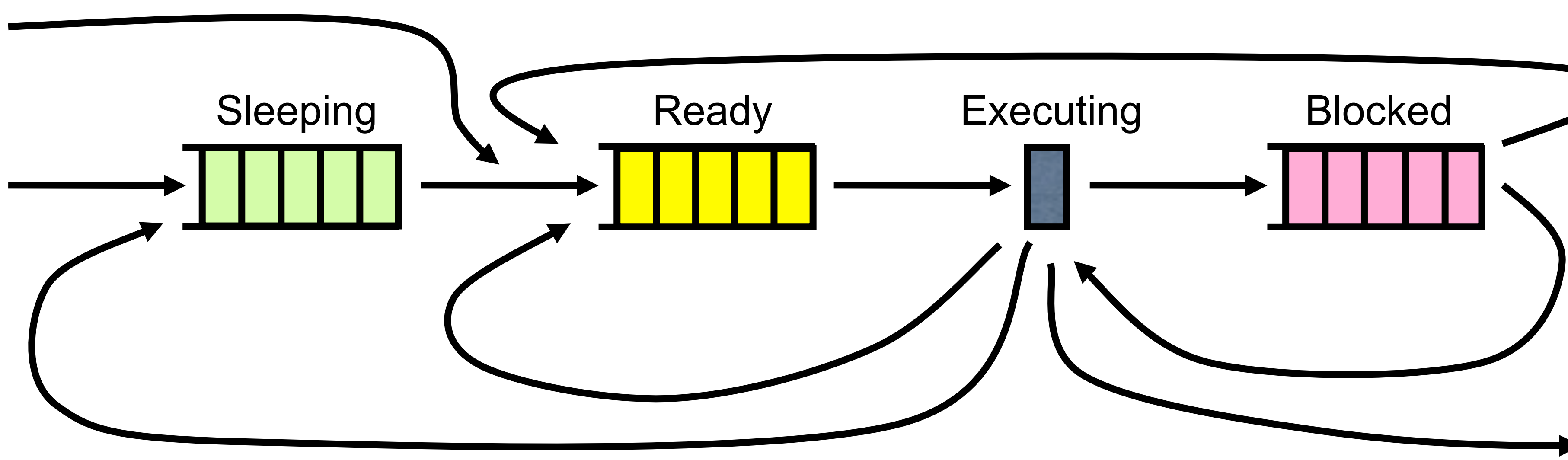




…realised as a set of queues

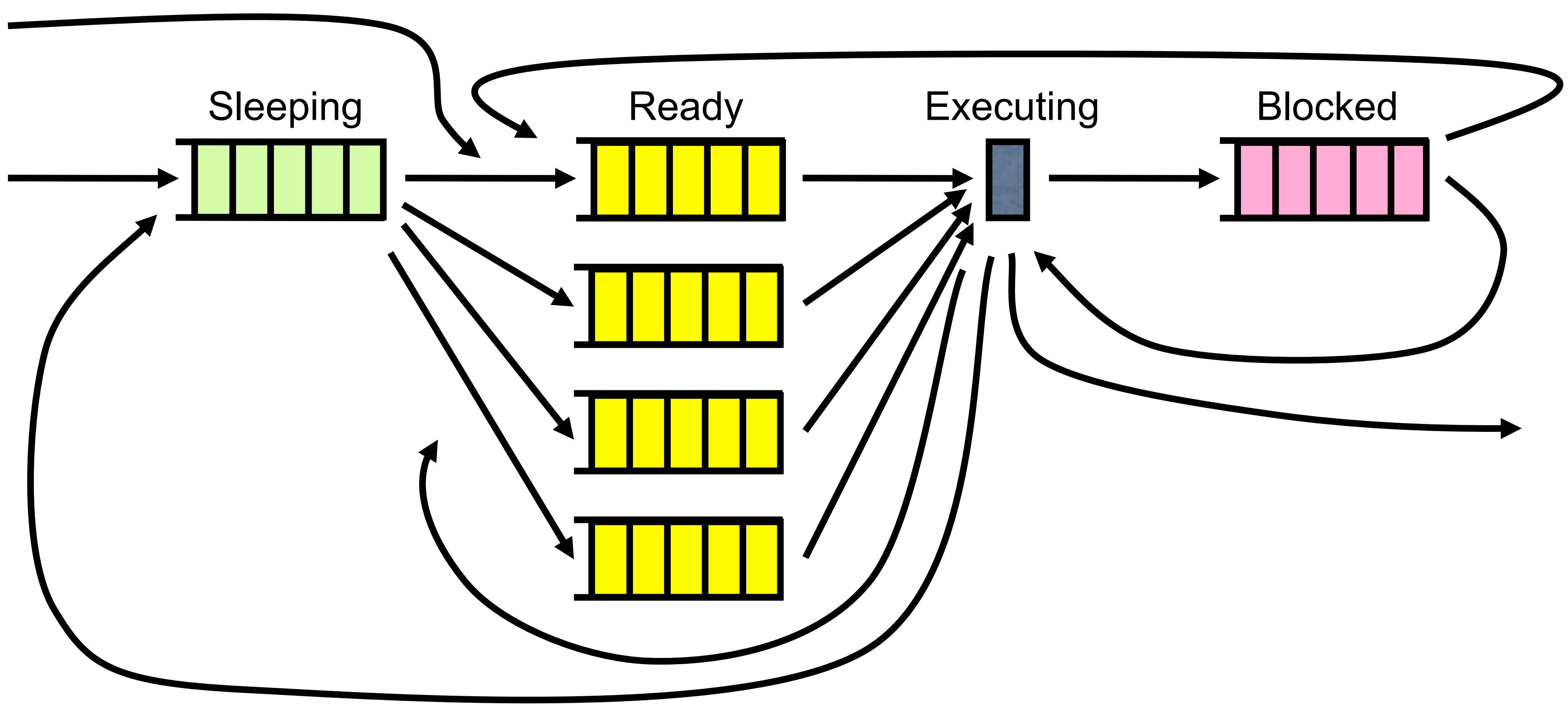
Building a Priority Scheduler

- How to schedule...
  - Periodic fixed priority tasks (RM and DM)?
  - Periodic dynamic priority tasks (EDF and LST)?
  - Sporadic and aperiodic tasks?
- Vary number of queues, the queue selection policy, service discipline
  - How to decide which queue holds a newly released thread?
  - How are the queues ordered?
  - From which queue is the next job to execute taken?



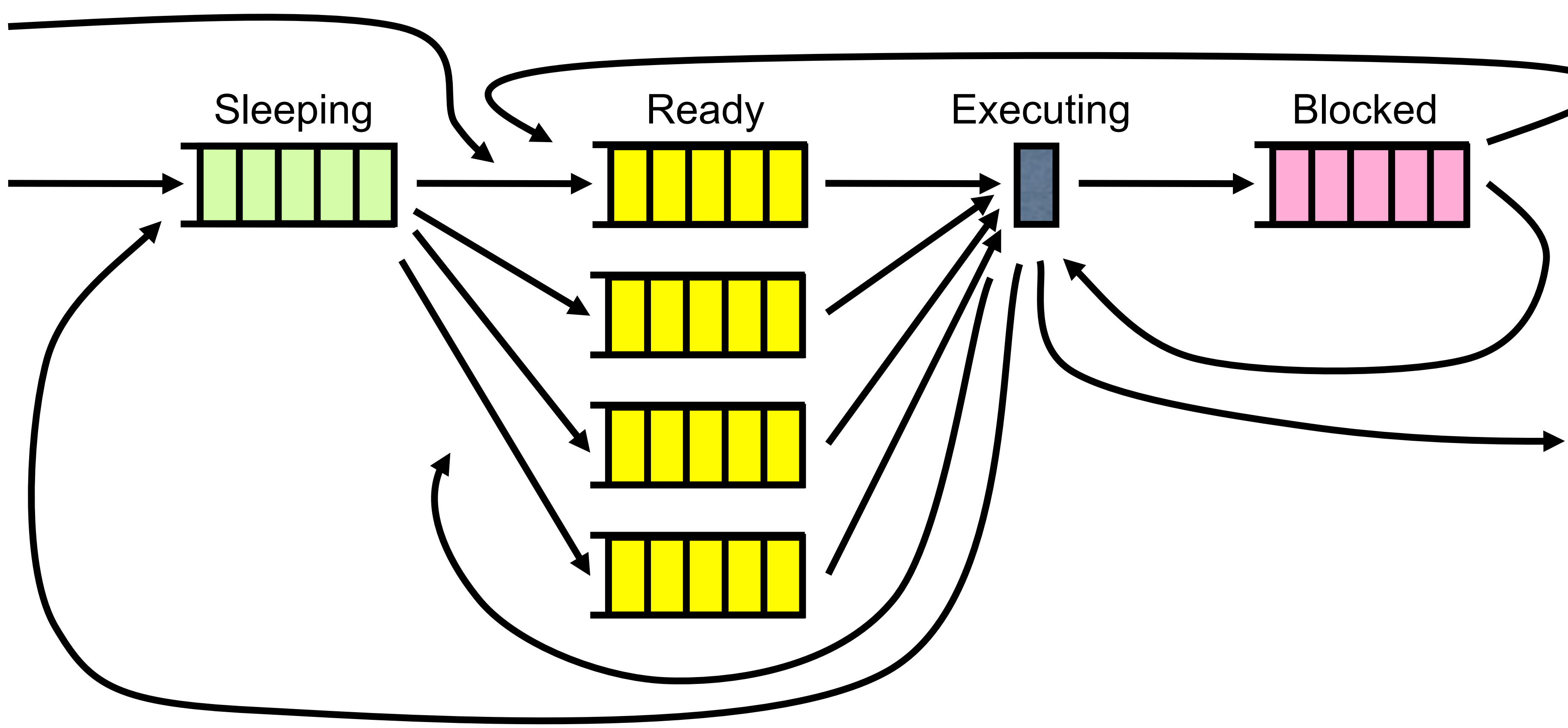
Fixed Priority Scheduling

- Provide one ready queue per priority level
  - Tasks inserted according to priority
  - FIFO or round-robin servicing
    - RR task budget depleted on each clock interrupt; yield when budget exhausted
    - FIFO tasks run until sleep, block or yield
  - Run task at the head of highest priority queue with ready tasks



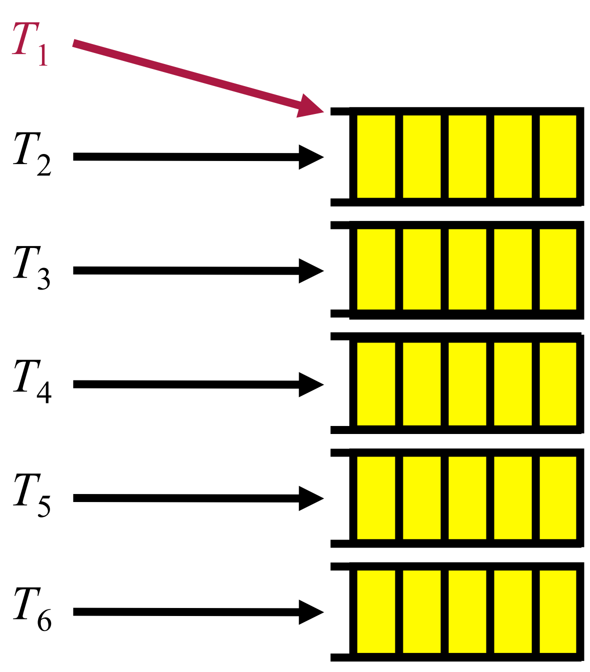
Fixed Priority Scheduling: RM

- Assign fixed priorities to tasks based on their rate $1/p$
  - Task resides in sleep queue until released at phase, φ
  - When released, task inserted into a FIFO ready queue
  - One ready queue for each distinct priority
  - Run task at the head of the highest priority queue with ready tasks



Practical Considerations: Limited Queues

- When building a rate monotonic system, ensure there are as many ready queues as priority levels
- May be limited by the operating system if present, and need more priority levels than there are queues provided



Implication: some tasks will be delayed relative to the "correct" schedule

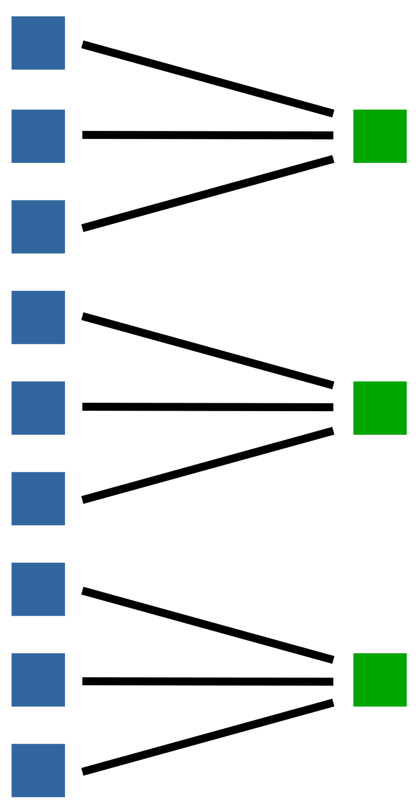
A set of tasks $T_E(i)$ is mapped to the same priority queue as task T_i

This may delay T_i by up to $\sum_{T_k \in T_e(i)} e_k$

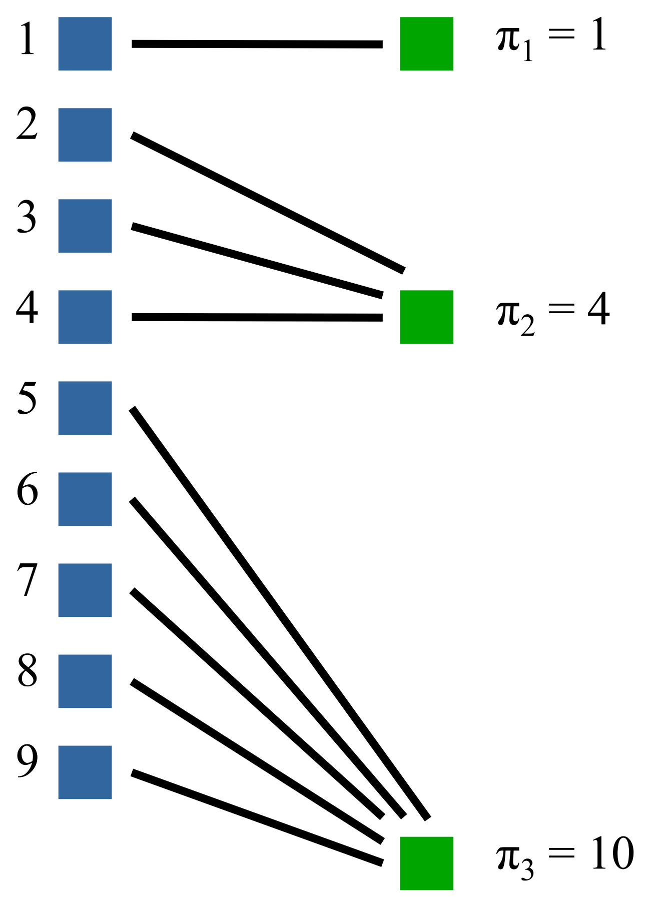
Schedulable utilisation of system will be reduced

Practical Considerations: Limited Queues

- How to map a set of tasks needing Ω_n priorities onto a set of Ω_s priority levels, where $\Omega_s < \Omega_n$?



Uniform mapping: $Q = | \Omega_n / \Omega_s |$ tasks map onto each system priority level

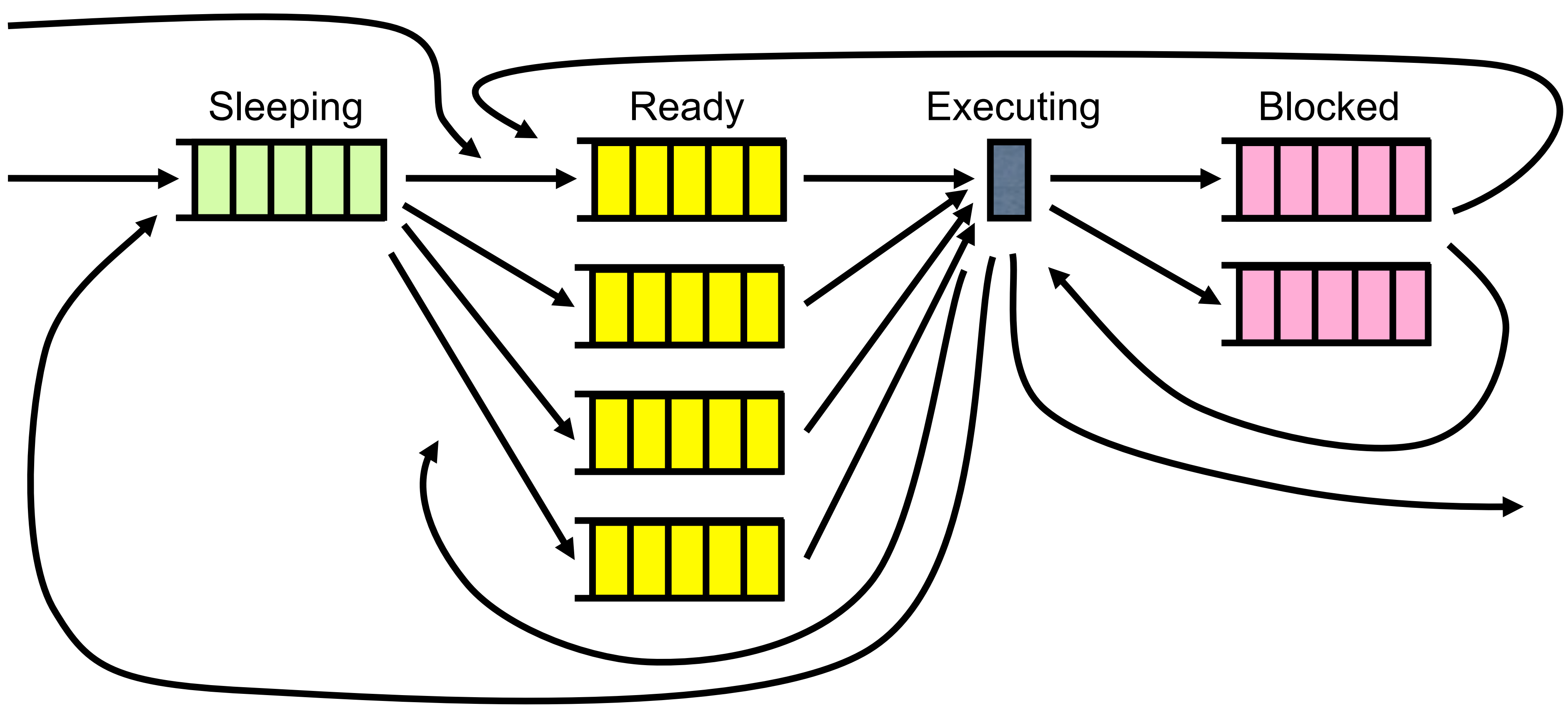


Constant Ratio mapping: $k = (\pi_{i-1}+1)/\pi_i$ tasks where k is a constant map to each system priority with weight, π_i

Better preserves executing time of high priority tasks.

Blocking on Multiple Events

- Tasks may block for many reasons
  - Disk I/O, network, inter-process communication, …
  - Use multiple blocked queues
- This is a RTOS typical priority scheduler



Dynamic Priority Scheduling

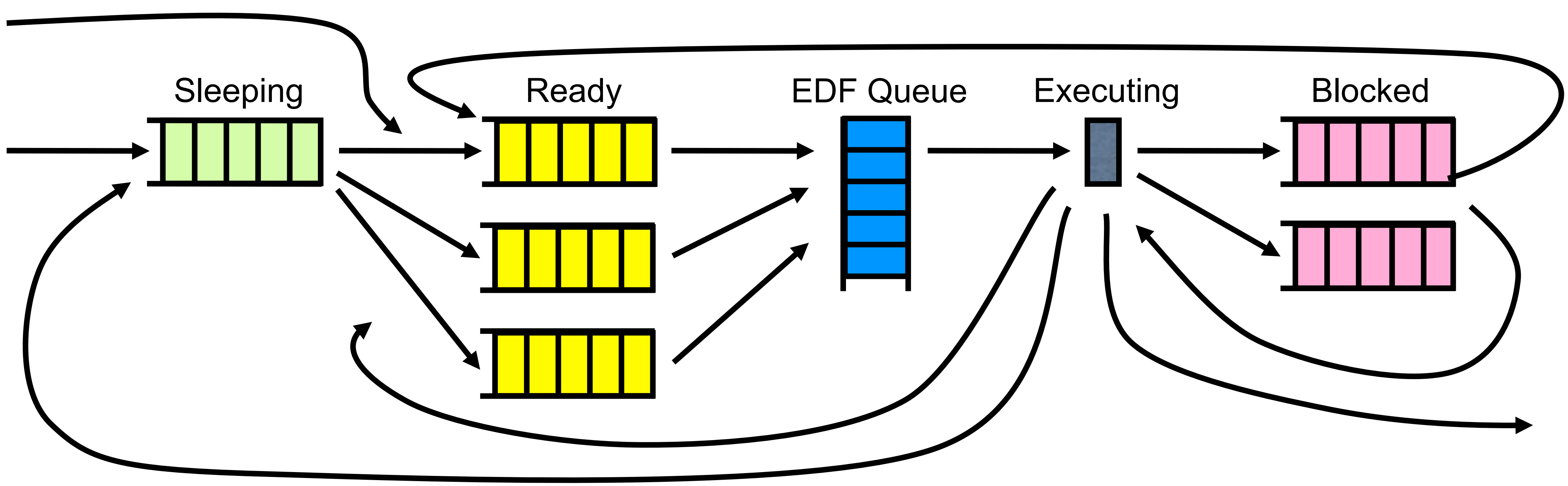
- Thread priority can change during execution
- Implies that threads move between ready queues
  - Search through the ready queues to find the thread changing it's priority
  - Remove from the ready queue; calculate new priority; insert at end of new ready queue
- Expensive operation:
  - O(N) where N is the number of tasks
  - Suitable for system reconfiguration or priority inheritance when the rate of change of priorities is slow
  - Naïve implementation of EDF or LST scheduling inefficient, since require frequent priority changes

Earliest Deadline First Scheduling

- To directly support EDF scheduling:
  - When each thread is created, its relative deadline is specified
  - When a thread is released, its absolute deadline is calculated from it's relative deadline and release time
- Could maintain a single ready queue:
  - Conceptually simple, threads ordered by absolute deadline
  - Inefficient if many active threads, since scheduling decision involves walking the queue of N tasks

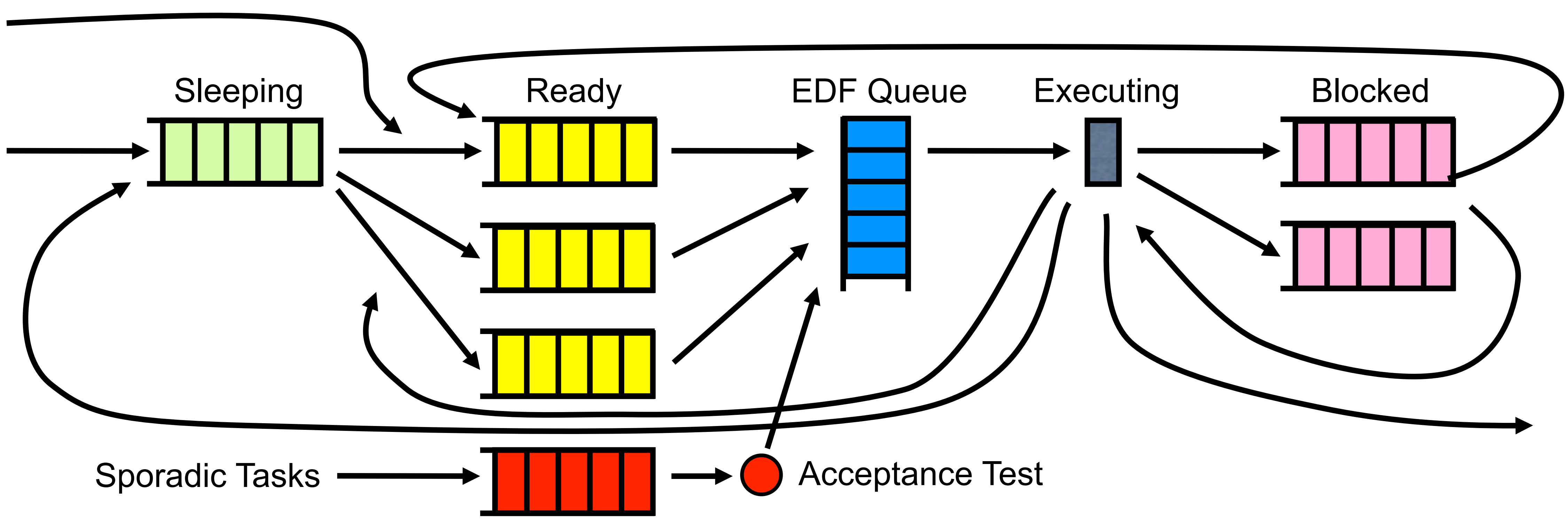
Earliest Deadline First Scheduling

- Maintain a ready queue for each relative deadline
- Maintain a queue, sorted by absolute deadline, pointing to tasks at the head of each ready queue
  - Updated when tasks complete; when tasks added to empty ready queue
  - Always execute the task at the head of this queue
  - More efficient, since only perform a linear scan through active tasks



Scheduling Sporadic Tasks

- Straight-forward to schedule using EDF:
  - Add to separate queue of ready sporadic tasks on release
  - Perform acceptance test
  - If accepted, insert into the EDF queue according to deadline
- Difficult if using fixed priority scheduling:
  - Need a bandwidth preserving server



Bandwidth Preserving Servers

- Server scheduled as a periodic task
- When ready and selected to execute, given scheduling quantum equal to the current budget
  - Runs until pre-empted or blocked; or
  - Runs until the quantum expires, sleeps until replenished
- At each scheduling event in the system
  - Update budget consumption considering:
    - time for which the bandwidth preserving server and other tasks have executed; algorithm depends on BP server type
  - Replenish budget if necessary
  - Keep remaining budget in the thread control block
  - Fairly complex calculations, e.g., for sporadic server

Summary

- Implementing real time systems
  - Key concepts and constraints
  - System architectures:
    - Cyclic executive
    - Microkernel with priority scheduler
- Implementing scheduling algorithms
  - Jobs, tasks, and threads
  - Priority scheduling of periodic tasks
    - Rate monotonic
    - Earliest deadline first
  - Priority scheduling of aperiodic and sporadic tasks