Scheduling in the Supermarket

Consider a line of people waiting in front of the checkout in the grocery store.

In what order should the cashier process their purchases?

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Scheduling Criteria

CPU utilization – Keep the CPU as busy as possible.

Throughput – Maximize the number of tasks completed per time unit.

Response time – Minimize the time required to finish a task.

Fairness – Try to give a similar amount of CPU time to similar tasks; avoid starvation.

Here, a *task* might be a thread, a single CPU burst in a thread, or an application-level service request (e.g., HTTP request).

The Nature of Program Execution

A running thread typically alternates between CPU bursts and I/O bursts (or I/O waiting times).

During a CPU burst, the thread is executing instructions.

During an I/O burst, the thread is waiting for the hardware and not executing any instructions.

Preemptive vs. Non-Preemptive

A non-preemptive scheduler only runs when the currently running thread gives up control of the CPU, by

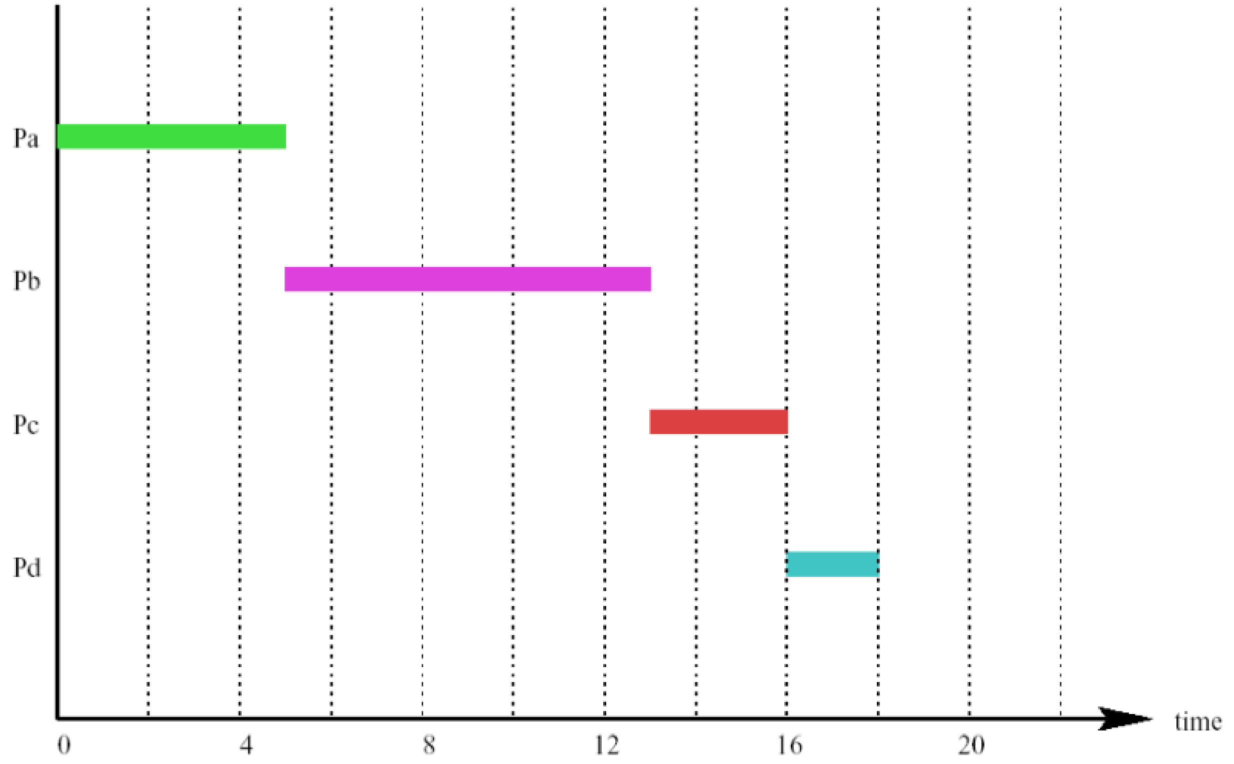
- terminating;
- blocking due to I/O activity;
- performing a Yield system call;
- performing some other system call and thus transferring control of the CPU to the kernel.

A preemptive scheduler may force a running thread to stop temporarily. The most important mechanism in a preemptive scheduler is the timer interrupt.
A preempted thread is inserted into the scheduler's *ready* queue.

FCFS Scheduling

First-Come, First-Served (FCFS) scheduling:

- non-preemptive: each thread runs until it blocks or terminates;
- scheduler maintains a FIFO ready queue.



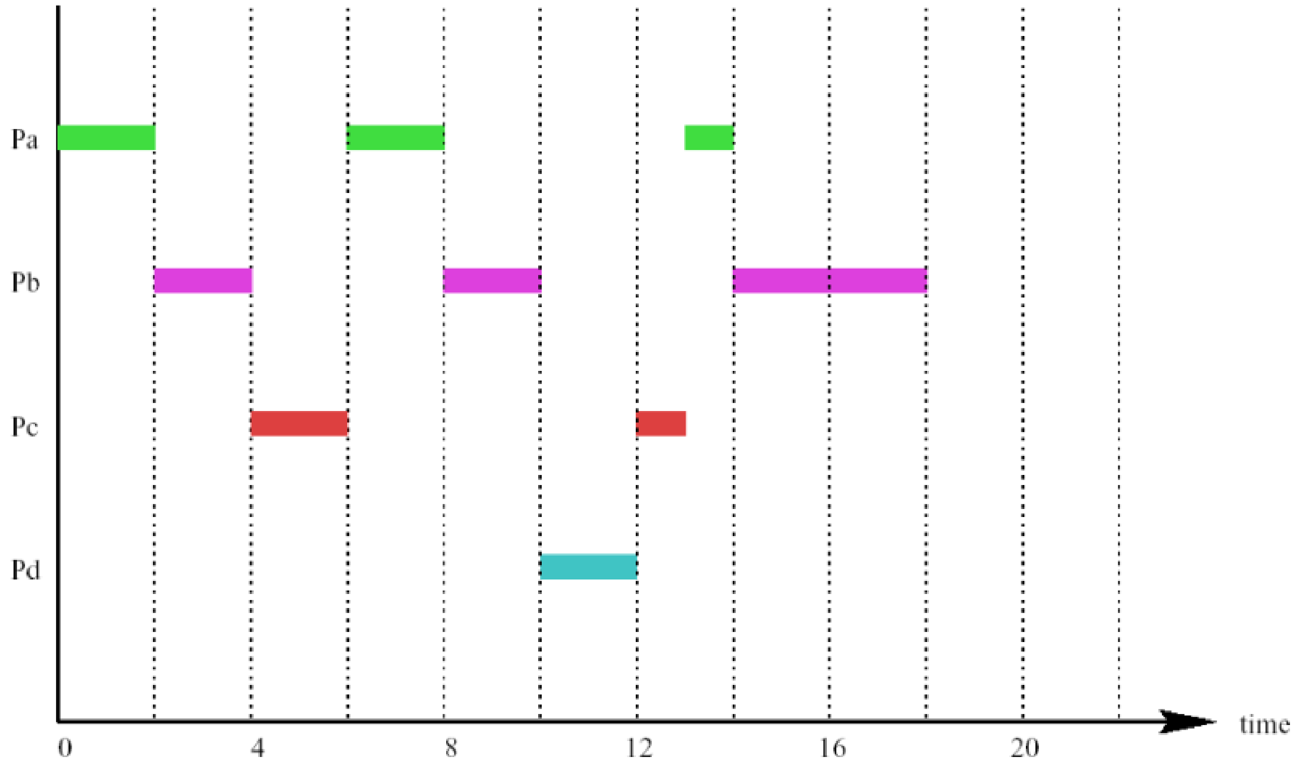
Initial ready queue: Pa = 5 Pb = 8 Pc = 3

Thread Pd (=2) "arrives" at time 5

Round-Robin Scheduling

Round-Robin is a preemptive version of FCFS

- running thread is preempted after a predefined sched. quantum;
- preempted thread is appended to the FIFO ready queue.



Initial ready queue: Pa = 5 Pb = 8 Pc = 3 Quantum = 2

Thread Pd (=2) "arrives" at time 5

Shortest Job First

The Shortest Job First (SJF) scheduling policy is a non-preemptive scheduling policy.

Ready threads are scheduled according to the length of their next CPU burst; the thread with the shortest burst is scheduled first.

SJF minimizes mean waiting time, but can lead to starvation.

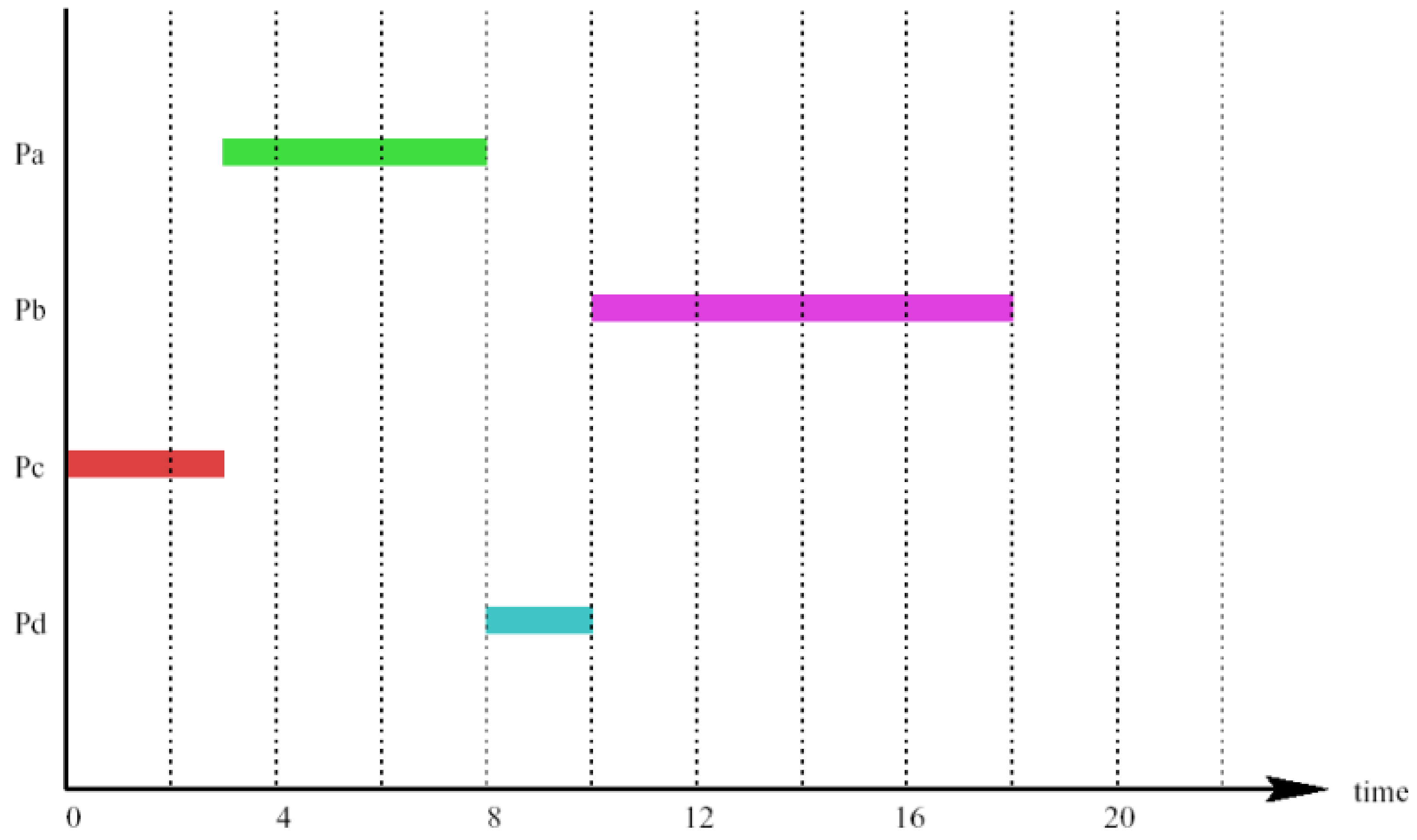
Requires knowledge of the future. Possible solution: Estimate the length of the next burst based on previous bursts.

$$B_{i+1} := \alpha * b_i + (1 - \alpha) * B_i$$

where B_i is the estimated length of the *i*-th burst, and b_i is its actual (measured) length. α is a dampening factor ($0 \leq \alpha \leq 1$).

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Shortest Job First



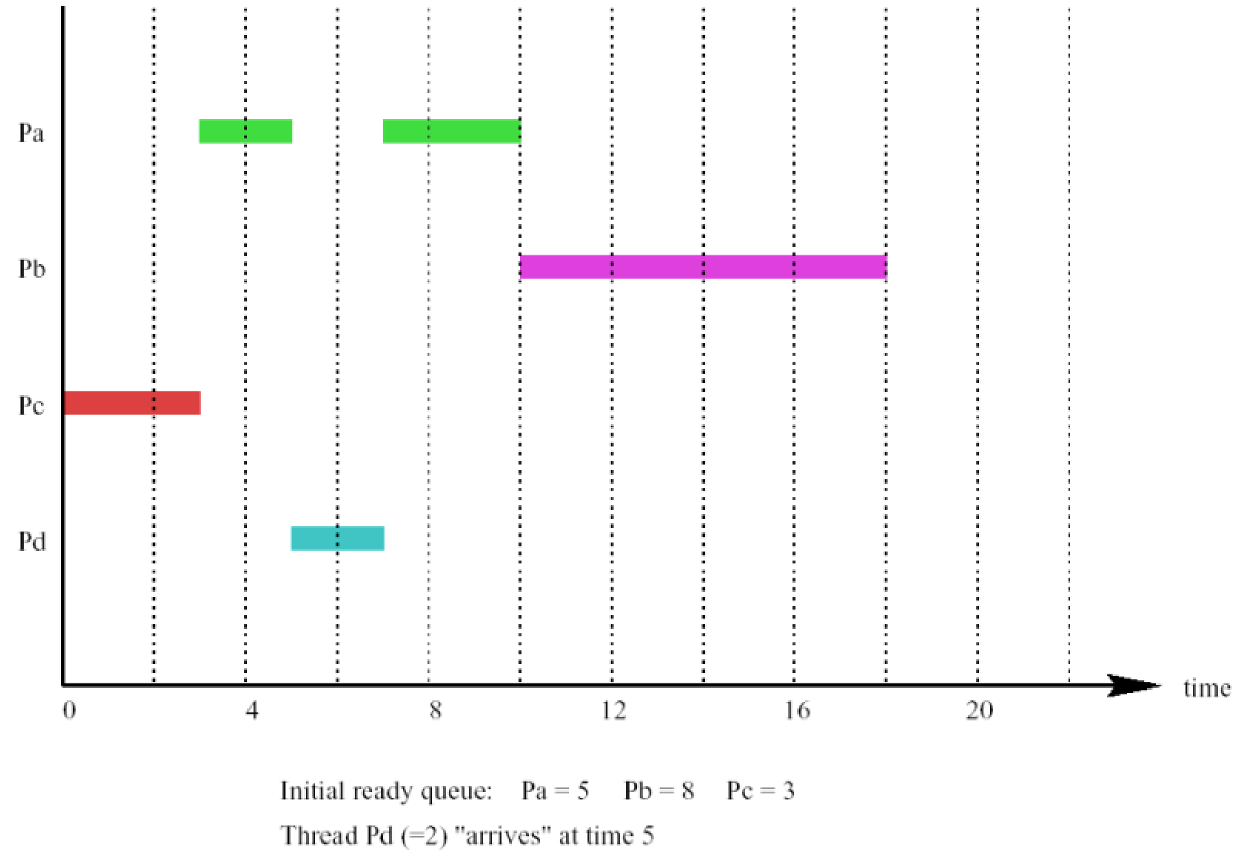
Initial ready queue: Pa = 5 Pb = 8 Pc = 3

Thread Pd (=2) "arrives" at time 5

Shortest Remaining Time First

Shortest Remaining Time First (SRTF) is a preemptive variant of the SJF scheduling policy.

Preemption may occur whenever a new thread enters the system (via Fork or ThreadFork).



Stefan Buettcher
<sbuettch@uwaterloo.ca>

Highest Response Ratio Next

Highest Response Ratio Next (HRRN) is a non-preemptive scheduling policy that takes the responsiveness of a thread into account.

Responsiveness: How often does a thread yield control of the CPU because it is blocked on I/O acitivity?

The response ratio of a thread *T* is defined as:

$$RR(T) = (w + b) / b,$$

where *b* is *T*'s CPU burst time, and *w* is its waiting time.

The scheduler chooses the thread with highest response ratio.

HRRN's goal is to keep the system responsive by picking “nice” threads. This only makes sense for non-preemptive scheduling.

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Probabilistic Scheduling

In probabilistic scheduling, CPU time slices are not assigned deterministically, but according to some probability distribution.

An example of probabilistic scheduling: *Lottery Scheduling*.

In lottery scheduling, each thread is assigned a certain number of lottery tickets. The scheduling decision is based on the outcome of randomly selecting a ticket.

The number of tickets a thread receives depends on certain thread characteristics, such as its interactivity.

Probabilistic scheduling can be used to approximate a deterministic scheduling strategy, but avoids starvation (probably...).

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Priorities

The scheduler might take priorities into account when making a scheduling decision.

Priorities can be based on (among other things):

- user ID (e.g., give higher priority to `root`);
- application type (e.g., kernel threads vs. user threads);
- explicit priority levels (e.g., `SetPriority` in Nachos).

The scheduler will:

- always prefer higher-priority threads over lower-priority threads;
- use a scheduling heuristic (e.g., Round-Robin) to schedule threads at the same priority level.

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Priorities

Priority levels can be:

- **static –** a thread always has the same priority;
- **dynamic –** the priority of a thread may change over time.

A general problem with priority-based CPU scheduling is that low-priority threads risk starvation.

This can be avoided with dynamic priority levels, where the priority of a thread is changed depending on how long it has been waiting for the CPU.

Multi-Level Feedback Queues

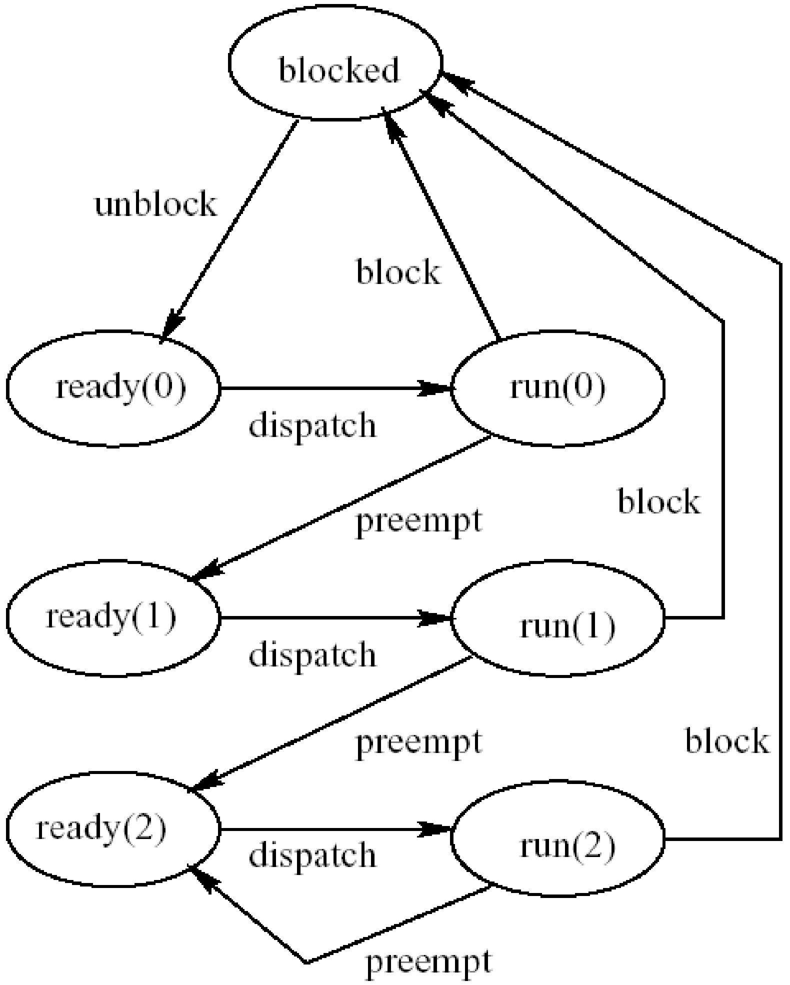
One way to change thread priorities dynamically is to use a multi-level feedback queue.

Properties of CPU scheduling with feedback queues:

- Priority is given to interactive threads.
- The scheduler maintains several *ready* queues; as long as there are threads in queue i, $i \leq j$, the scheduler never selects a thread from queue j.
- When a thread gets unblocked (or created), it is put into ready queue 0.
- When a thread from queue i gets preempted, it is put into queue $i+1$.

Multi-Level Feedback Queues

State diagram for a 3-level feedback queue



Different Types of Interactivity

So far, we have considered blocking as evidence that a thread is interactive (interacting with the hardware or interacting with other threads).

Interactive threads ought to receive higher priority.

However, there are different types of interaction.

In a desktop environment, user interaction is more important than interaction with the hard drive.

How can we find out whether a thread is interacting with the user?

Summary of Scheduling Algorithms

FCFS

+ Simple to implement; low overhead; no starvation.
- Gives poor response time for interactive processes.

Round-Robin

+ No starvation; reduced waiting time variance; good response time for interactive processes.

SJF (Shortest Job First), SRTF (Shortest Resp. Time First)

+ Optimal average response time.
- Effectiveness depends on the accuracy of estimating the burst lengths; starvation is possible.

Feedback Queues

+ Good response time for interactive processes.
- CPU-intensive processes might starve.

Multi-Processor Scheduling

When a computer has more than 1 CPU, scheduling becomes more complicated.

Two different paradigms:

- **Asymmetric multiprocessing –** Have the different CPUs in the system perform different tasks; e.g., one CPU is responsible for I/O tasks, the other CPU does numerical computations.
- **Symmetric multiprocessing (SMP) –** All CPUs basically do the same job.

Issues in symmetric multiprocessing:

- **Load balancing –** Keep both CPUs similarly busy (or not!).
- Differentiate between real CPU cores and virtual cores (hyperthreading).

Multi-Processor Scheduling

More issues in symmetric multiprocessing:

CPU affinity – Let the user (or the operating system) decide which CPU a given thread to run on. This is important because it can lead to better utilization of the CPU cache.

If a process consists of more than one thread, should the threads be executed on the same CPU or on different CPUs?

In general, everything becomes more complicated, because of true parallelism (as opposed to mere concurrency).

Operating system providers/vendors usually supply two different versions of their kernel, one for single-CPU systems, one for SMP.

Scheduling Scopes

Covered so far: Short-term scheduling ("What task to run next?").

Medium-term scheduling:

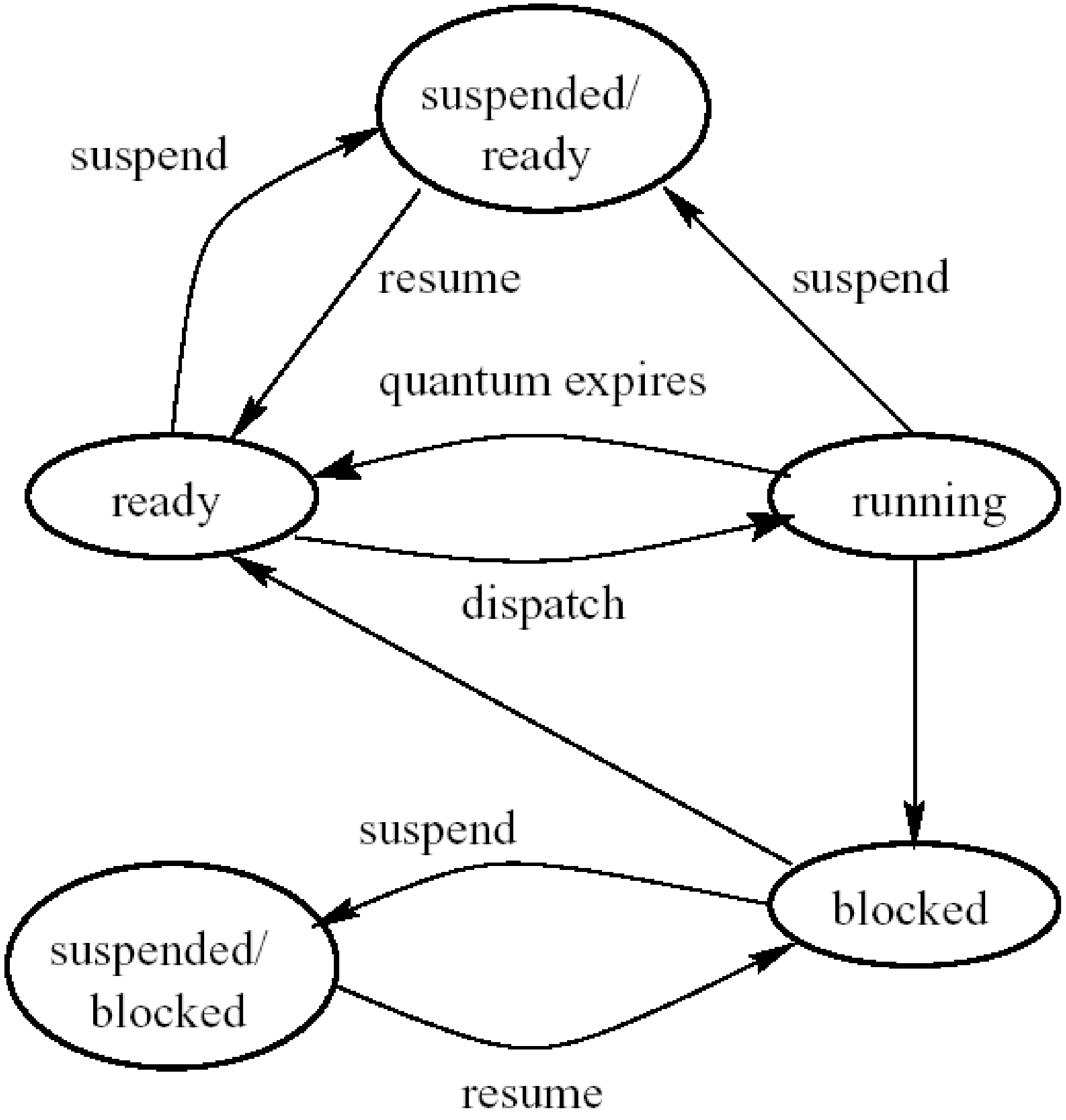
- Suspension and resumption of processes.
- Processes are usually suspended because the system is overloaded (typically main memory).
- Suspended processes may release some resources (e.g., memory, CPU), but not all (e.g., open file descriptors).
- Sometimes, the operating system allows users to suspend a process (UNIX: Ctrl+Z).

Long-term scheduling:

- Usually happens outside the core operating system.

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Scheduling States with Suspend/Resume



Stefan Buettcher
<sbuettch@uwaterloo.ca>

CPU Scheduling in Solaris

Each task in Solaris can be in one of four classes:

1. Real-time
2. System
3. Time sharing
4. Interactive

Real-time tasks have the highest priority, followed by *system* tasks.

Within each class, priority levels can be assigned to different tasks. Priority levels are changed via a multi-level feedback queue.

A higher priority results in a shorter time quantum.

If there are multiple threads at the same priority level, Solaris uses a Round-Robin scheduling algorithm to dispatch them.

Stefan Buettcher
<sbuettch@uwaterloo.ca>

CPU Scheduling in Linux

Linux assigns tasks a priority between 0 and 140:

0..99: real-time priority levels
100..140: non-real-time (*nice* levels)

A higher-priority thread is assigned a longer time quantum.

Linux maintains a list of all runnable threads (the *runqueue*, 1 per CPU). A runqueue consists of two arrays: *active* and *expired*. The *active* array contains a tasks with remaining time in their time slices; the *expired* array contains all tasks with no time left.

The scheduler chooses the highest-priority task from the *active* array until all threads are in *expired*. Then the arrays are swapped.

Thread priorities can be changed (+/- 5) based on interactiveness whenever a thread it is moved from *active* to *expired*. Interaction includes keyboard, mouse, hard drive, sound card.

CPU Scheduling in Windows XP

Windows XP supports 32 priority levels:

0: special system process
1..15: real-time processes
16..31: variable priority

Each priority level has its own *ready* queue.

Real-time processes are always scheduled first.

When a task gets preempted, its priority is lowered (numerical priority is increased), but stays within a class-specific limit.

When a thread becomes unblocked, its priority is raised; the amount of change depends on what it was waiting for (e.g., keyboard vs. hard drive).

Windows running in the foreground receive a higher priority (and approximately 3x the original time slice).

Stefan Buettcher
<sbuettch@uwaterloo.ca>