# Operating Systems

## Scheduling

Sven Dziadek

October/November 2022

EPITA

# Vocabulary

Recall:

- **Program** (passive):
  Static object containing code
- **Process** (active):
  Program in execution

**Task**

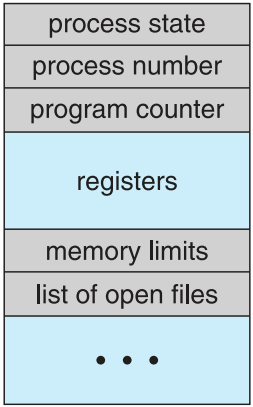
Process or thread in the context of scheduling

**CPU**

CPU, processor and core will be used indistinguishable

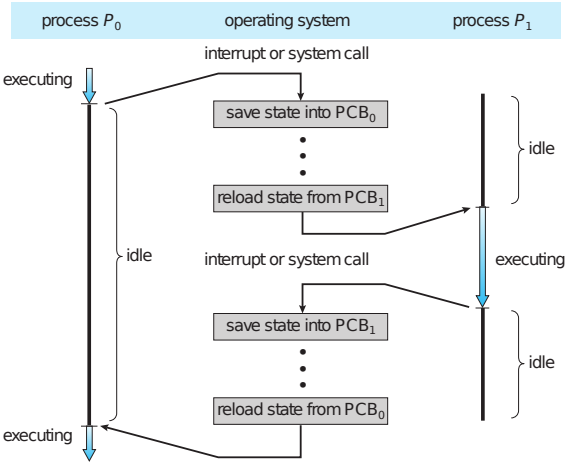
# Processes

# Process Control Block – Context




- Representation of processes in the Linux kernel:
  - `task_struct`
  - `PEB` on Windows; `struct task` on Darwin
- Contains all of the task state:
  - state (ready, running, waiting, ...)
  - scheduling information
  - memory mappings
  - process id, thread group id
  - registers (in `struct thread_info`)
  - PDBR (Page Directory Base Pointer)

# Context Switch




**Context Switch**

Change running process on CPU

Store and restore:

- CPU registers
- process state
- virtual memory space (e.g., CR3)

# Init (now mostly systemd)

**Init**

First process launched by the kernel on boot

You cannot kill init

Doing so results in a kernel panic

**History**:

`init` was a bunch of shell scripts tied together: `rc` then `sysvinit`.

**Now**:

most distros use `systemd`

# Daemons

**Daemon**

Linux service running in user mode

Names mostly ending on -d

`init` is responsible for launching daemons

`daemon(3)` detaches process from terminal to run in background

# Process Tree

- Processes organized in a tree
- `fork(2)` produces children
- New process gets a new `PID` (process id)
- If the parent dies before the children children are assigned to `init`

# Zombies

- Parent can ask its children's exit status
  $\Rightarrow$ processes are not automatically destroyed after their execution
- Only destroyed ("reaped") at end of execution after `wait(2)` or `waitpid(2)`

**Zombie**

Terminated process whose parent has not yet called wait

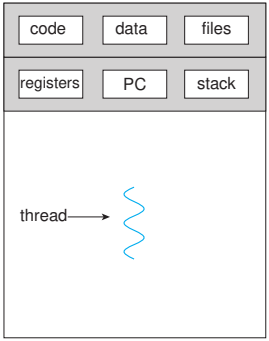
- No longer running
- Its context is not freed
- `init` regulary waits for its children

# Threads

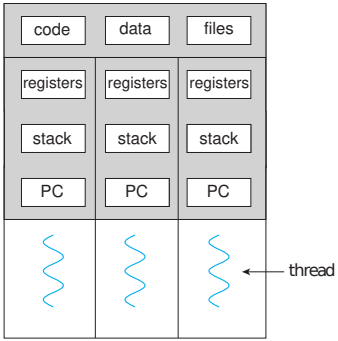
**Multi-threaded process**

Virtual memory space is shared among all threads

This makes creation and context-switching fast




single-thread




multi-thread

# Threads on Linux

**Lightweight processes (LWP)**

Threads are processes that share resources

Threads have the same process id but different LWPid

Create threads with `clone(2)` by giving `CLONE_THREAD` as flag

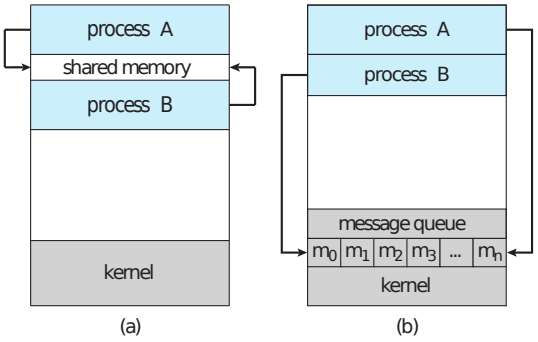
Better: use `pthreads` (POSIX threads) library

# IPC

# Inter-Process Communication (IPC)

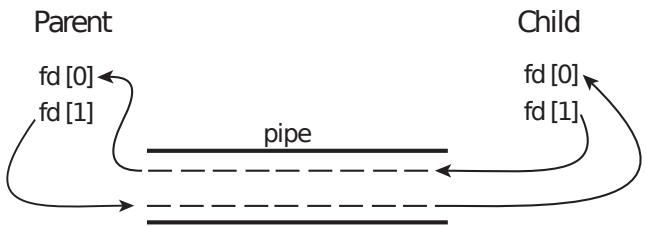
Cooperative processes
need to communicate:

- Shared memory (see memory lecture)
- Message passing



# Pipes

- Created with `pipe(2)`.
- Returns two file descriptors: a write end and a read end
- Unidirectional
- Read and writes occurs on a kernel managed buffer



# Named Pipes

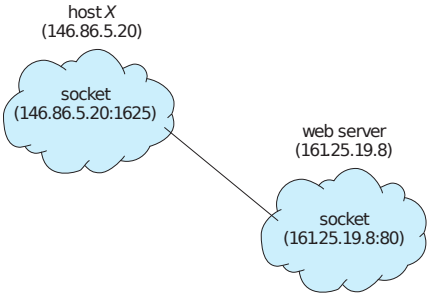
- Created with `mkfifo` (or `mknod`)
- Similar to anonymous pipes created with `pipe(2)`.
- Has a path in the virtual file system:
  - One process can open it for reading
  - One process can open it for writing

**Demo**

- Create named pipe (`mkfifo my_pipe`)
- Write to it (`echo "test" > my_pipe`)
- Read from it (`cat my_pipe` in different cmd)

# Unix Socket

- Created with `socket(2)` with `AF_UNIX` flag
- Can be anonymous or named, like pipes
- Similar to Internet sockets but local
  Handled by the kernel
- Many advantages compared to pipes:
  - More than two processes can communicate
  - Can send file descriptors through it



# Scheduling

# Scheduler: Overview

- Scheduler manages CPU time
- More tasks than available cores
- Some tasks block (e.g., for I/O completion like reading a file)
  - This task can't run in the meantime
  - Use CPU for another task!

**Goal**

Efficient use of CPU time

## Scheduling criteria

Different scheduling algorithms for different use cases

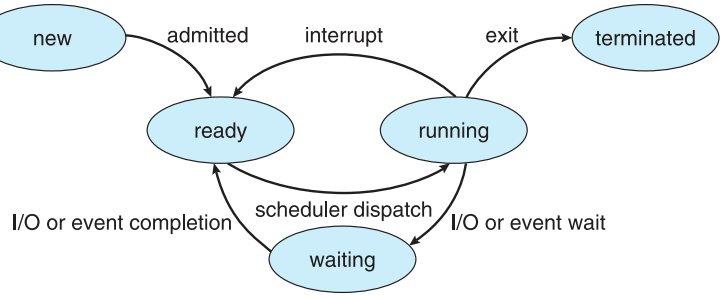
**Maximize**:

- CPU utilization
- Throughput

**Minimize**:

- Turnaround time
- Waiting time
- Response time

# Textbook state machine of scheduler



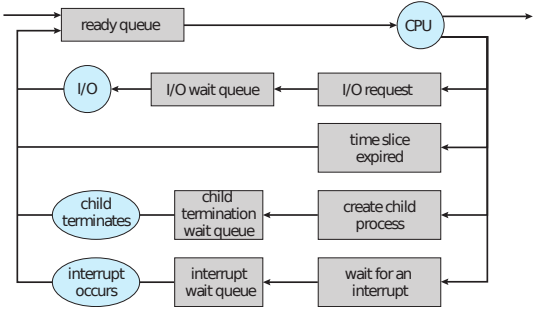

**Dispatcher**

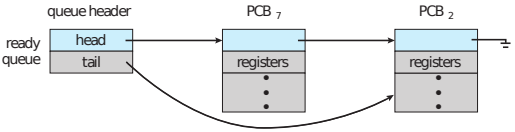
Sets up the processor to run a process

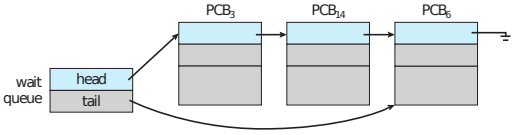
# Scheduling Queues

Data structures of the scheduler:

- Process table (with `task_structs`)
- Ready queue
- Wait queues



# When to schedule?

- This process: Running → Waiting
  Example: I/O request
- This process terminates

**Nonpreemptive scheduling**

Process yields autonomously

- New process spawns
- Other process: Waiting → Ready
  Example: I/O finished
- Other process: Running → Ready
  Example: Interrupt

**Preemptive scheduling**

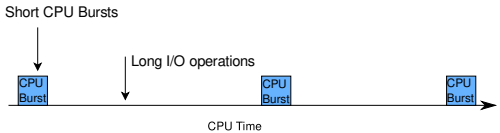
Processes can be interrupted at any time

Be aware of race conditions!

# Types of Tasks

## I/O bound

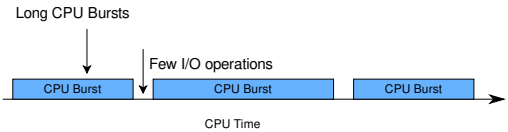
More I/O than computations




Give higher priority

## CPU bound

More computations than I/O




Give lower priority

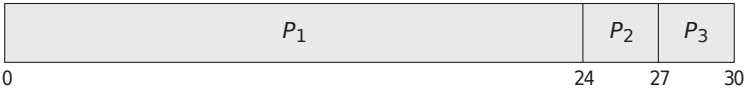
## Real-Time

Respect of deadlines & Predictability

# Scheduling Algorithms

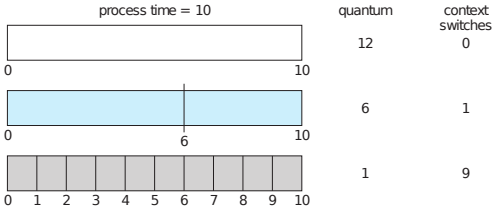
# FCFS: First Come First Served

- Pros:
  - Nonpreemptive
  - FIFO for ready processes
- Cons:
  - Nonpreemptive
  - Accumulation effect
- Bad for interactive system
- OK/Good for batch systems
- On Linux: `SCHED_FIFO`



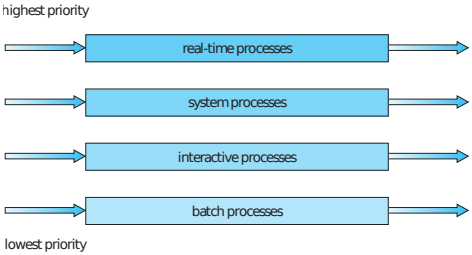
# Round-Robin

- Similar to FCFS
  but with added preemption
- fixed time quantum
- alternating tasks quicker
- time quantum:
  large $\Rightarrow$ FCFS
  small $\Rightarrow$ high context switch overhead
- A little bit better for interactive systems
- On Linux: `SCHED_RR`



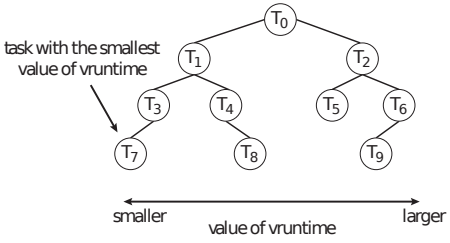
# Multiple Priority Queue

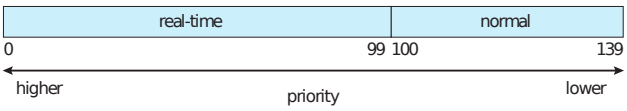
- Split tasks into multiple priorities
- Different scheduling policy for each priority
- Scheduling between the different priorities



# CFS: Completely Fair Scheduling

- Try to run each task at least once
  in fixed **targeted latency**
- Record **virtual run time** with fair clock
  $\Rightarrow$ higher priority = time elapses slower
  lower priority = time elapses faster
- Order processes by virtual run time
  in a red-black tree
- Privileges I/O-bound tasks
- Current Linux Scheduler: `SCHED_NORMAL`



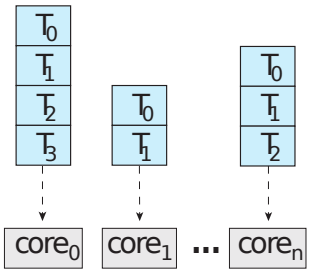
# Multi-Processor Scheduling

**Symmetric Multiprocessing (SMP)**

Each CPU is self-scheduling

Advantages over asymmetric multiprocessing:

- Private run queues prevent bottleneck
- More efficient use of cache memory




**Load Balancing**

Migrate tasks from CPU under high load to other CPUs

## Documentation

- `sched(7)`
- `sched_setscheduler(2)`
- `sched_getscheduler(2)`
- `sched_yield(2)`
- `SCHED_FIFO`: First-in First-out scheduling
- `SCHED_RR`: Round-robin scheduling
- `SCHED_DEADLINE`: Sporadic task model deadline scheduling
- `SCHED_OTHER`: Default Linux time-sharing scheduling
- `SCHED_BATCH`: Scheduling batch processes
- `SCHED_IDLE`: Scheduling very low priority jobs