NTNU
Norwegian University of
Science and Technology

Lecture 8: Operating Systems

Asbjørn Djupdal
ARM Norway, IDI NTNU
2013

Lecture overview

- Operating systems
- Boot loader and boot process
- Program loader
- Processes and scheduling
- Inter-process communication
- OS Power management

Operating System Tasks

- Booting
- Process scheduling
- Memory management
- Hardware abstraction
- Power management
- Filesystems, network stacks, …
- ...

Do you need a proper OS?

- Depends on your project
- Simple microcontroller system, with a single execution thread:
  - Probably just as good to write your own runtime-system
- Complex system with multiple threads, memory management, network stack etc:
  - Go for a proper OS

Making your own

- You need startup code
  - Boot vectors
  - Exception handling
  - Stack and heap
  - Cache and MMU
  - Jump to main()
- You probably want support for libc
  - You must implement support for the syscalls required by libc.
  - Can then use malloc(), strncmp(), printf(), etc
- You probably need support for critical sections
  - Interrupt handlers communicating with main loop
  - mutex, semaphore
- A simple task scheduler can be useful

Lecture overview

- Operating systems
- Boot loader and boot process
- Program loader
- Processes and scheduling
- Inter-process communication
- OS Power management

The boot loader

- Located such that CPU automatically enters the boot loader on power-on
  - Many systems have several boot-loader stages
- Basic HW initialization
- Reads OS kernel from permanent storage (Flash, SD-card, HD)
- Sets up OS parameters and jumps into OS kernel

Das U-Boot

- One of the most used boot loader for embedded systems
- Easy to port to new HW
- Supports reading OS kernel from flash or from the network
- Command line over RS-232 serial terminal
- Can script the boot process

Kernel initialization

- Initializes cache and virtual memory
- Initializes kernel structures
- Sets up and initializes HW platform and drivers
- Loads OS userspace init code and executes it

Linux kernel initialization

- Gets machine info from boot loader
  - Machine type, amount of memory, kernel parameters
  - New systems: FTD (Flattened Device Tree) describing available HW devices
- Kernel decompressing
  - Kernel is usually compressed to save storage space
- Kernel boot
  - Initializes platform according to given machine type
  - Initializes drivers according to given FDT
- Mounts root file system
- Usually loads and runs /sbin/init

User space init, Unix

- /sbin/init
- Several variants:
  - BusyBox
  - SysV init
  - Upstart
- BusyBox
  - Common for embedded systems
  - Runs /etc/inittab
- Upstart:
  - Ubuntu default init
  - Event based
  - Starts services in /etc/init/ in response to system events
  - One of these services should start the main application or user interface

User space init, Android

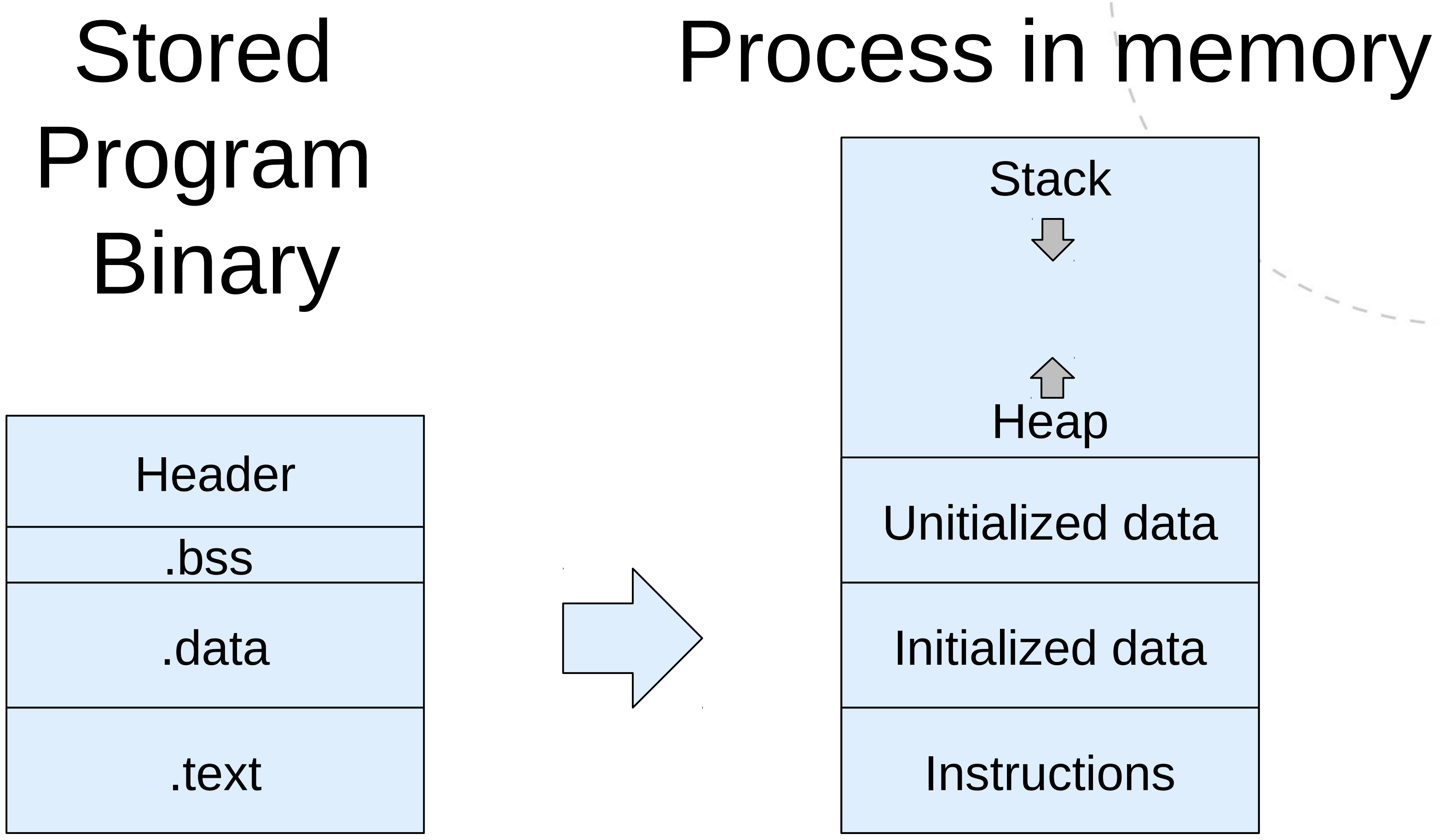
- Starts services in init.rc
  - Start zygote service
    - Dalvik VM (Java virtual machine)
    - Preloads and initializes core library classes
    - Start systemserver
      - Starts all the system services, including UI
- Android native applications are normal linux programs
  - But lacks most of the libraries
- Normal Android applications are java programs
  - Application process is a clone of Zygote
    - Saves startup time and reduces memory footprint

Lecture overview

- Operating systems
- Boot loader and boot process
- Program loader
- Processes and scheduling
- Inter-process communication
- OS Power management

Application loading

- The loader loads the stored executable file into memory
  - Checks permissions
  - Copies program image into memory
  - Handles program arguments
  - Jumps to program entry point
- A loaded program is called a *process*
- Some systems need a relocatable loader
  - Loads to arbitrary base address
  - Pointers are absolute and not relative to load address
- Systems with dynamic libraries need a dynamic linker as part of the load process


Stored
Program
Binary
Process in memory
Stack
Heap
Header
.bss
.data
.text
Unitialized data
Initialized data
Instructions

Shared libraries

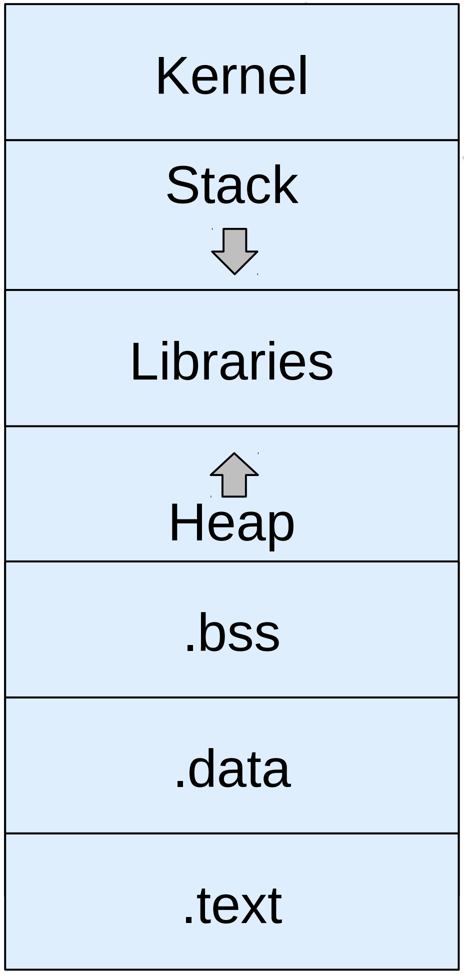
- Static libraries: Linked into the application at compile time
- Shared libraries: Linked dynamically when application loads (.so, .dll)
- Typically exists in the address space of the application process
- The application code has compiled in stubs that calls the dynamic linker at startup and handles communication with the library

Unix shared libraries

- ld.so : dynamic linker
- Where to find shared libraries?
  - Path specified in executable
  - ld.so built-in search path
  - User search path: LD_LIBRARY_PATH
- `ldd`: Examine which dynamic libraries a program needs

Linux application memory map

- `cat /proc/XXXX/maps`



Lecture overview

- Operating systems
- Boot loader and boot process
- Program loader
- Processes and scheduling
- Inter-process communication
- OS Power management

Process

- A *process* is an instance of a program in memory
  - Instructions and state
- Single-tasking operating systems only have one process in memory at a time
- Multitasking operating systems can have several processes in memory
  - All modern operating systems
  - The *scheduler* manages which process is running
- Each process has an associated data structure used by the process scheduler

Processes and threads

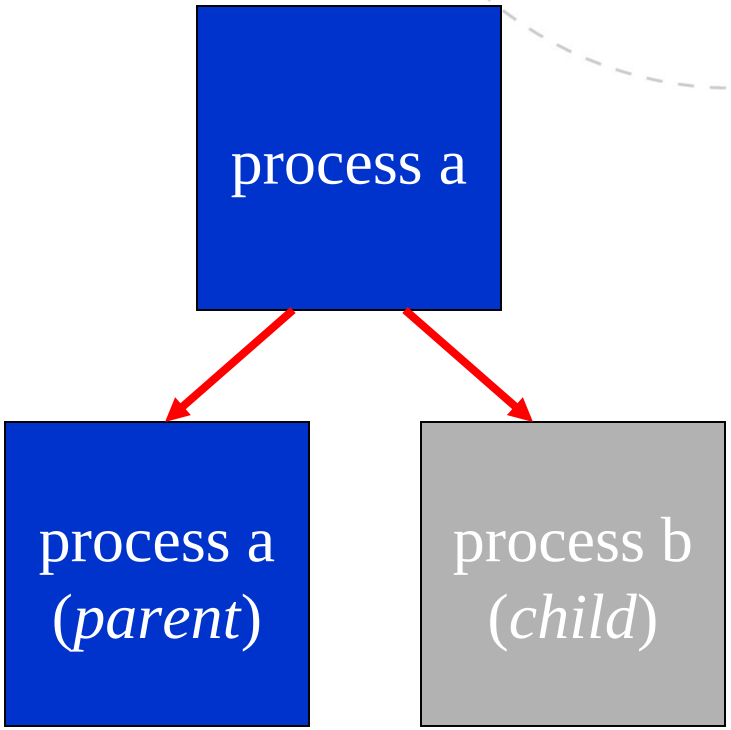
- Each process has its own address space
  - Communication between processes go through kernel
    - Inter-Process Communication (IPC)
- A process can have several threads
  - Typically a light-weight process, associated with the main process
  - All threads run in the same address space
    - Share variables, etc.

Why multiple processes and threads?

- Helps handle complex timing
- Asynchronous input
  - User interface in one thread
  - Calculations in separate worker thread
- Multi-rate systems (I/O with different speeds)
  - sound, video, network

POSIX processes

- Standard API for unix-like systems
  - Linux, *BSD, OSX, NT-kernel, ...
- A process is started with fork():
  - The parent process continues to execute the original program
  - The child process starts execution at this location in the original program



fork() and execve()

- execve(): replaces current process with new
  - Runs the loader
- fork(): creates a child process
  - Creates a copy of the parent process

```
childid = fork();
if(childid == -1) {
  /* error */
} else if (childid == 0) {
  /* child operations */
} else {
  /* parent operations */
}
```

Posix threads

- Extension to support threads in posix-systems
- pthread_create()
- Implementation defines how threads are realized in the system
- Linux: NPTL
  - Each thread is handled like processes by the kernel, except sharing memory

Context switching

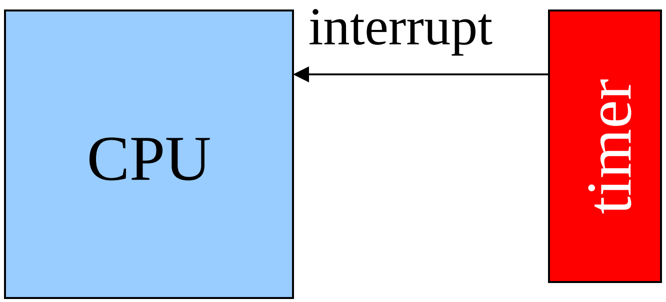
- Terms:
  - Context: The process state needed to be remembered when switching processes
    - Contains register state and kernel structures
  - Context switch:
    - Removes the running context and stores it
    - Inserts the new context
- Change which process is running on the CPU
- Implementation decisions:
  - Who decides when to switch?
  - How is the context switch implemented?
- Two variants:
  - Cooperative multitasking
  - Preemptive multitasking

Cooperative multitasking

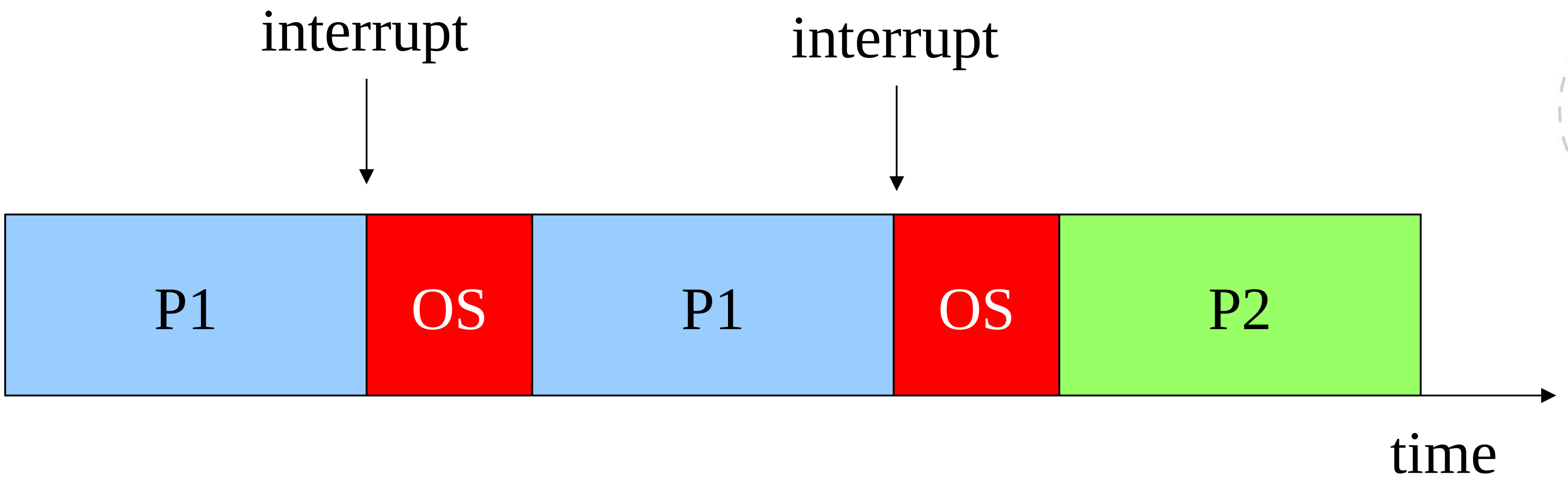
- A process runs until it decides to give up control
  - Either explicit: yield()
  - Or implicit: Blocking sys-call
- The scheduler decides which process to run, after a process yields.
- Assumes cooperation: A process can prevent others from getting CPU time

Preemptive multitasking

- OS decides when context switches are carried out and how processes are scheduled
- A timer interrupt gives control to the CPU at regular intervals



Preemption control flow

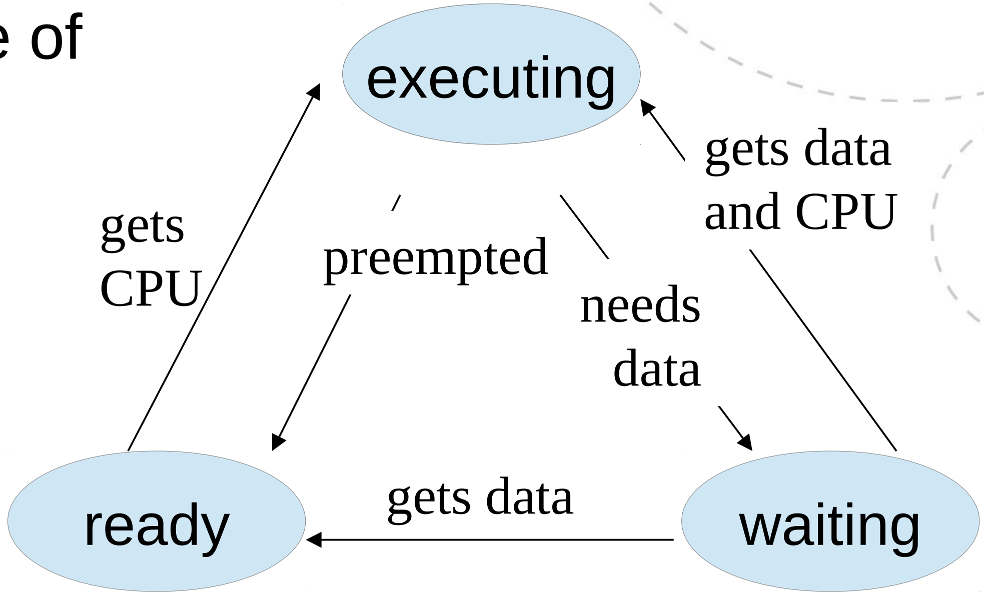


Preemptive context switch

- Timer controlled interrupt gives control to the OS
- OS stores the context and kernel state of the interrupted process
- OS chooses the process that should run next
  - Scheduling algorithm
- OS loads the context and kernel state of the new process
- The new process starts execution

Process states

- A process can be in one of three states
  - Executing: running on CPU
  - Ready: waiting for CPU
  - Waiting: waiting on I/O



OS bookkeeping

- Need to keep track of
  - Process priorities
  - State of all processes
  - Context and kernel structures of suspended processes
- Processes can be defined
  - Statically
    - Keep process information in static structures
    - Only suitable for purpose buildt embedded systems
  - Dynamically
    - Allocate data structures dynamically

Priority based scheduling

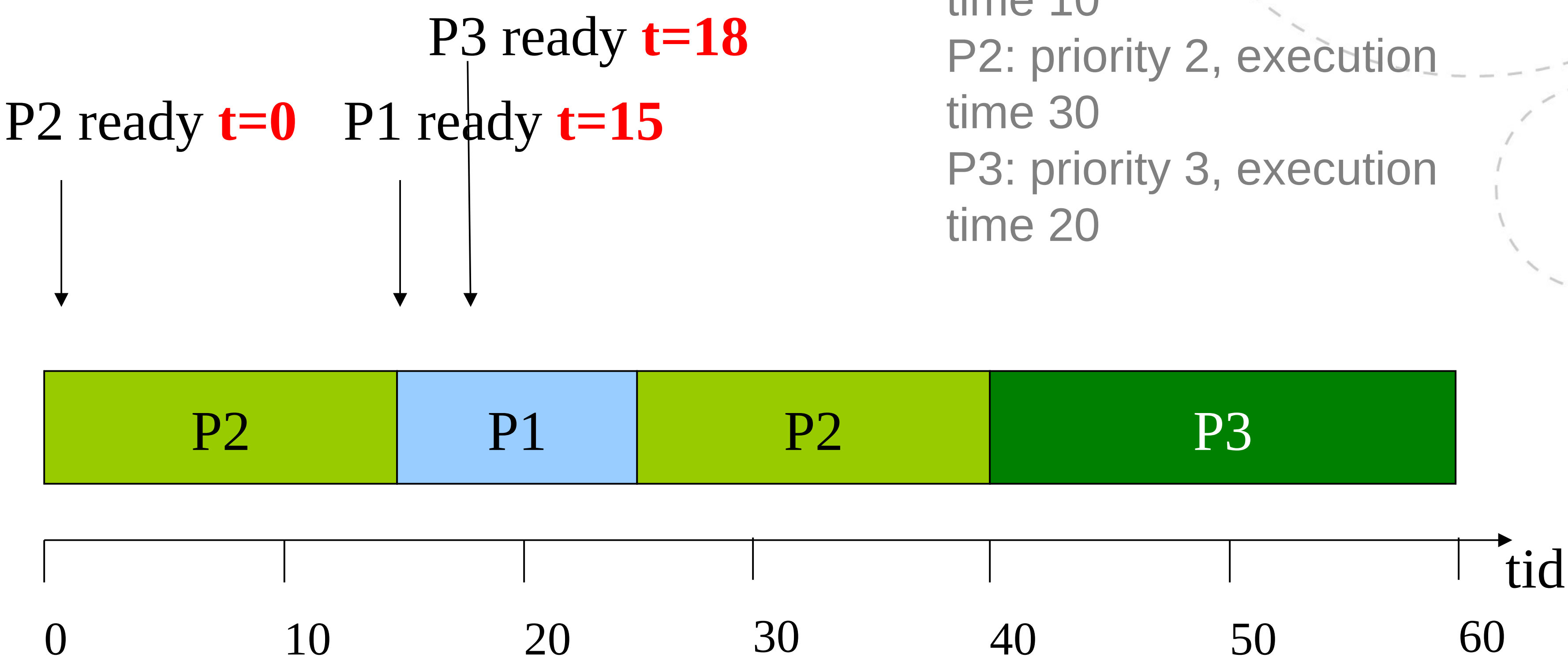
- Each process has a priority
- The highest priority ready process is allowed to run on the CPU
- Priorities can be:
  - Static
  - Vary over time
- POSIX:
  - nice / renice
  - Gives priority from -20 (max) to 19 (min)

Priority based scheduling example

- Rules:
  - Each process is assigned a static priority (1 = highest)
  - The highest priority ready process is granted the CPU
  - The process runs until its finished or goes into the wait state
- Processes
  - P1: priority 1, execution time 10
  - P2: priority 2: execution time 30
  - P3: priority 3: execution time 20

Priority based scheduling example

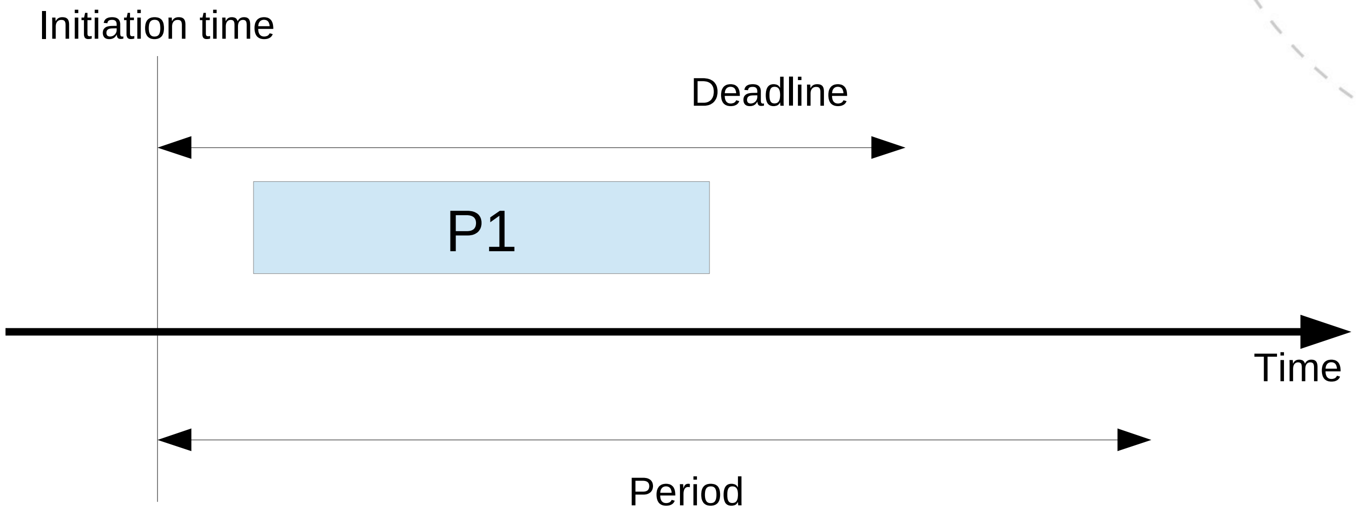
P1: priority 1, execution time 10
P2: priority 2, execution time 30
P3: priority 3, execution time 20



Real time scheduling

- Workstations try to be fair and maximize CPU usage
  - Access to CPU in proportion to process priority
  - Avoid starvation
- Embedded systems often have hard deadlines
  - Must give high priority processes access, even if low priority processes starves
  - Example: Engine control, spark signal must come at the correct time

Deadlines



- Initiation time
- Deadline
- Period

Example scheduling algorithms

- Round robin
  - Cycles between each ready process
  - Fairness
  - Can not guarantee completion time, response time increases with number of processes
    - Not suitable for real time deadlines

Example scheduling algorithms

- Rate monotonic scheduling
  - All processes are periodic on a single CPU
  - Static scheduling policy
  - Always runs highest priority ready process
  - Assumes all deadlines at end of periods
  - Assigns highest priority to process with shortest period
  - Rarely achieves 100% utilization

Example scheduling algorithms

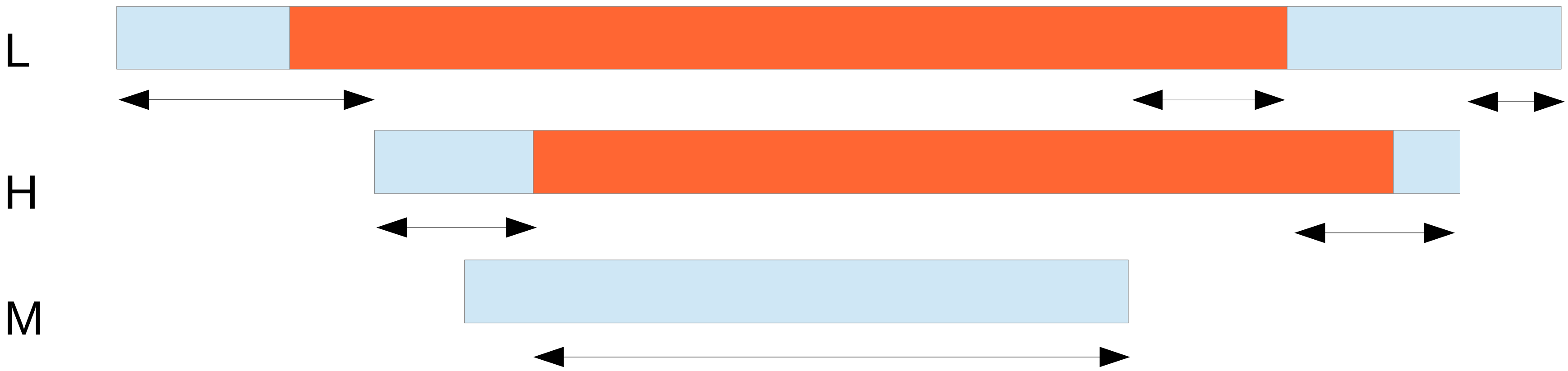
- Earliest deadline first scheduling
  - Dynamic scheduling policy
  - Always runs processes with the closest deadline
    - Need to keep processes sorted on dynamically chaning deadlines
  - Can achieve 100% utilization

Priority inversion

- A lower priority process can block a higher priority process by occupying a shared resource
  - E.g an I/O device

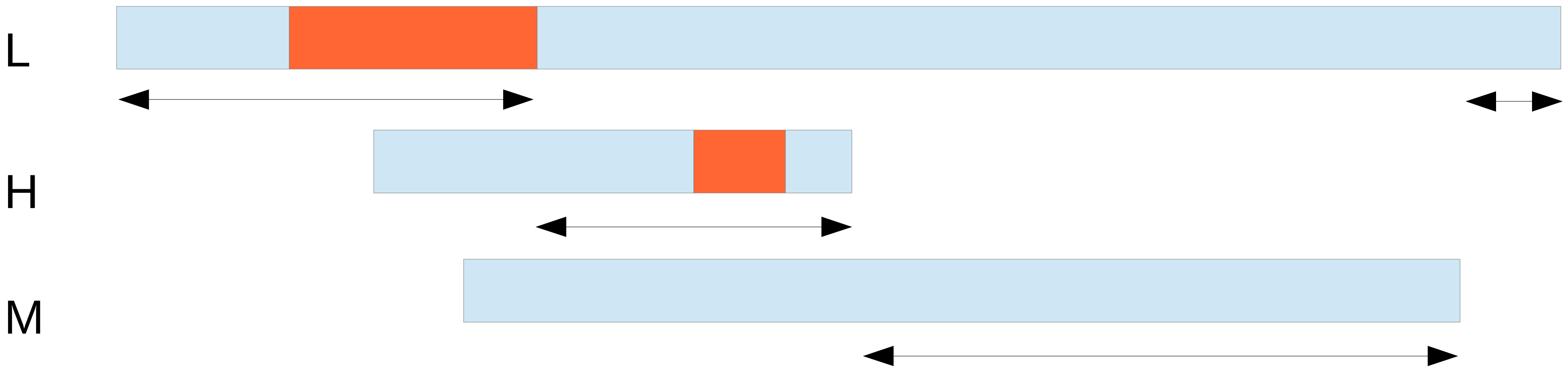
Priority inversion

- Scenario:
  - Three processes: L, M, H (low, medium, high priority)
  - L takes resource R
  - H wants resource R, but must wait until R is released
  - M preempts L and runs, even though H has higher priority
  - Result: H is blocked until M is done



Priority inversion

- Solution: Temporarily promote the priority of a process when it requests a shared resource
  - Makes the process run until it has finished using the resource
  - Process priority is then returned to the default
  - Complicates static analysis of real time systems



Lecture overview

- Operating systems
- Boot loader and boot process
- Program loader
- Processes and scheduling
- Inter-process communication
- OS Power management

IPC: Interprocess communication

- Processes have separate address spaces
  - Kernel must offer mechanisms for IPC
- Main types:
  - Signals: Simple signals between processes
  - Message passing: Processes communicate over an explicit communication channel
  - Shared memory:
    - Several processes can share the same physical memory

Signals

- Unix mechanism for simple communication between processes
- Similar to an interrupt
  - One process can signal another
  - The receiving process' signal handler is then executed, acting on the signal
- POSIX signals:
  - SIGABRT, SIGTERM, SIGFPE, SIGILL, SIGKILL, SIGUSR1, SIGUSR2
- C support: `signal.h`
  - `sigqueue(), kill()`
- Unix shell: `kill <pid>`

Message based communication

- Pipes
  - $ cat file.txt | grep "searchtext"
  - Handled as files in POSIX
  - Create new pipe: pipe()
  - A parent process can create a pipe before fork()-ing, to communicate with child
- Named pipes
  - Like normal pipes, but visible as a fifo-file in the file system
  - Can be opened and accessed by any process
  - mkfifo()
- Message queues
  - Can post messages to a queue without having to wait for a receiver
  - Possible to give messages priority

Shared memory

- OS can setup a memory region mapped into the address space of several different processes
- Can then communicate through this shared memory
- POSIX: shm_open(), mmap()

Race conditions

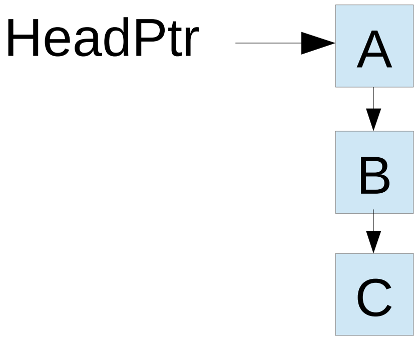
- Can arise if two threads try to access the same shared resource at the same time
  - Example: Add element to a linked list

  - Process 1 reads list pointer
  - Process 2 reads list pointer
  - Process 1 attaches old list pointer to element and updates list pointer
  - Process 2 attaches old lister pointer to element and updates list pointer
  - Result: Element of process 1 is lost

  - Conclusion: We need mechanisms for handling atomic operations

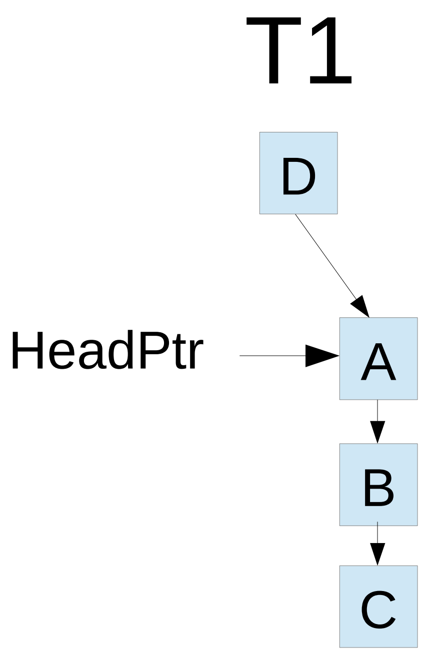
Race conditions

- Can arise if two threads try to access the same shared resource at the same time



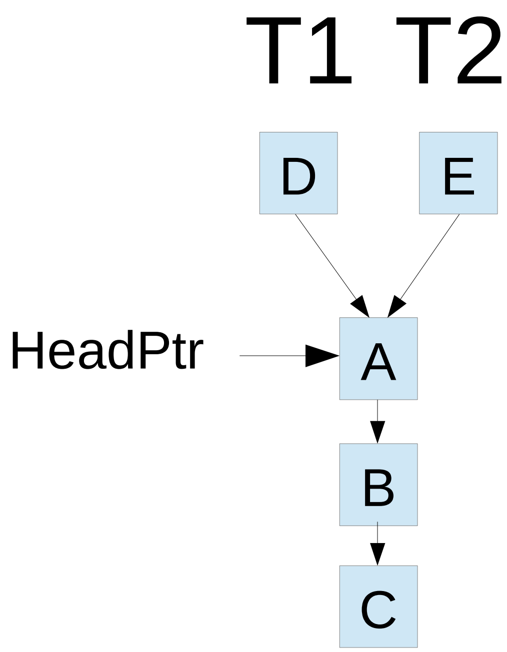
Race conditions

- Can arise if two threads try to access the same shared resource at the same time



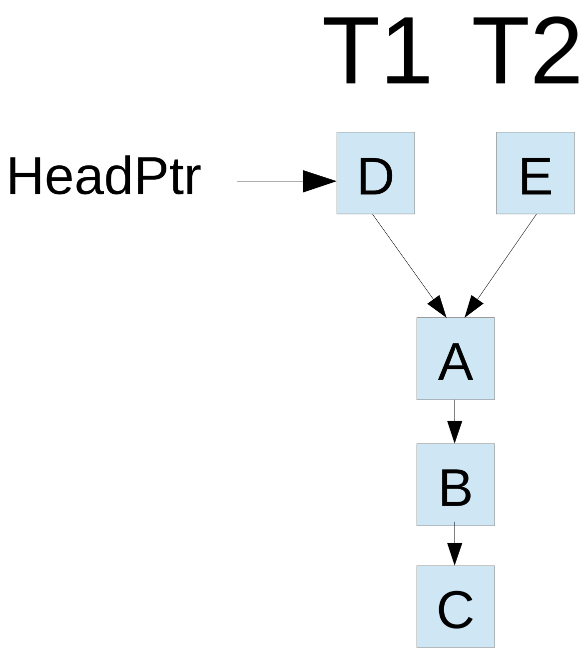
Race conditions

- Can arise if two threads try to access the same shared resource at the same time



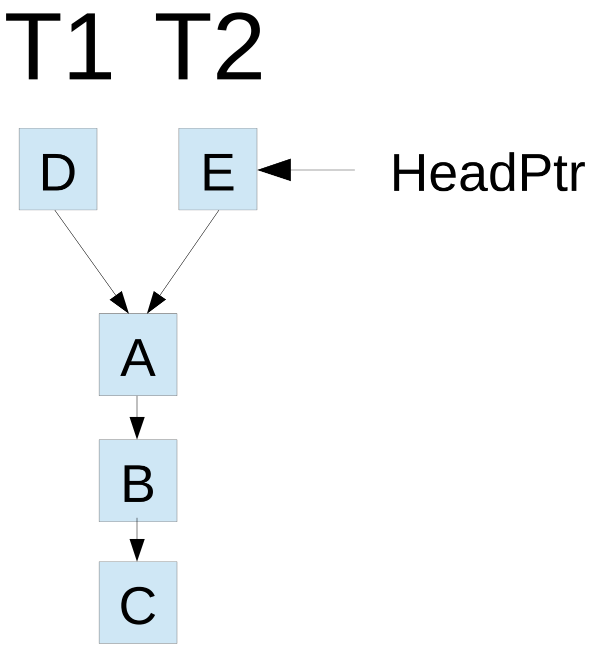
Race conditions

- Can arise if two threads try to access the same shared resource at the same time



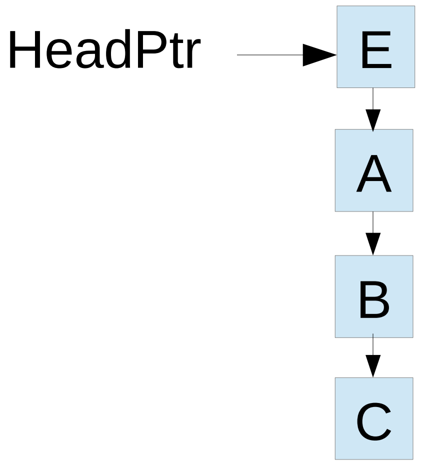
Race conditions

- Can arise if two threads try to access the same shared resource at the same time



Race conditions

- Can arise if two threads try to access the same shared resource at the same time



Synchronization mechanisms

- Critical section
  - Code accessing a shared resource that must not be interleaved with other threads accessing the same resource
  - Access to critical section must be atomic
  - Examples:
    - Accessing shared memory
    - Accessing an I/O device
- OS must provide semaphores or mutexes
  - Protect critical sections
- Protocol:
  - Lock the mutex with P()
    - Only one can have access
  - Execute operations on shared resource
  - Release the mutex with V()

Implementing mutex

- Need to atomically test and set a variable
- Single processor systems:
  - Possible to get atomicity by temporarily disabling interrupts
- Multi-processor systems: Need HW support
  - ARM: LDREX, STREX instruction pair

```
TAKEN = 0xFF

    mov  r1, #TAKEN
try:
    ldrex    r0, [LockAddr]
    cmp r0,  #0
    strexeq  r1, r0, [LockAddr]
    cmpeq r0, #0
    bne try
```

Lecture overview

- Operating systems
- Boot loader and boot process
- Program loader
- Processes and scheduling
- Inter-process communication
- OS Power management

Power management

- Power management: Controlling system resources with the aim of reducing power consumption
- OS can prioritize for power consumption
  - Similarly as it does for execution time
- Main techniques:
  - Sleep modes
  - Run-time enabling/disabling devices
  - Frequency and voltage scaling

Power management

- Power consumption and performance are often conflicting goals
- Power management is not without overheads
  - Entering sleep modes costs time and energy
  - Exiting from sleep modes costs time and energy
- Remember: The saving of entering a sleep mode must outweigh the overheads

Sleep modes

- When the CPU is idle, it can be put to sleep to save energy (powered off)
- Several sleep modes are often available, with various compromises regarding energy consumption and wake-up time
- Typical variations:
  - Which clocks are running
  - Which I/O controllers are powered on
- OS must support this by entering appropriate sleep modes in the idle loop

Power management strategies

- When to perform power management operations
- Request-driven: Wake up when a request arrives
  - Gives a delayed response
- Predictive shutdown: Attempt to predict the time to the next request
  - Common case: Already running when the request arrives
  - Delayed response when prediction is wrong
  - Works well with system where request timing can be predicted with high accuracy (e.g. time slot based communication)

Avoiding wake-ups

- Kernel:
  - Tickless idle: Do not call the scheduler periodically while idle
  - Calculate time until next scheduled process and sleep until then
  - Wake up earlier in case of unexpected events
  - Avoid polling kernel threads
- User space:
  - Avoid processes that polls (does an action on periodic timers)

Dynamic voltage and frequency scaling (DVFS)

- Reducing clock frequency reduces dynamic power consumption
- Reducing voltage reduces both dynamic and static power consumption
- Given HW that can do DVFS, the OS should adjust both of these such that the CPU is running “just fast enough”.

Power management in Linux

- Suspend
  - User space forces system to sleep
- cpuidle
  - Architecture dependent idle loop (enter sleep modes)
- Tickless idle loop
- cpufreq
  - Handle DVFS
- Runtime power management
  - Turn off unused devices while running

powertop

Drivers and power management

- Drivers must participate to make power management work
- Each driver must register callback functions for doing suspend, resume and runtime power management

Common embedded operating systems

- Hard real time
  - FreeRTOS
  - QNX
  - VxWorks
  - WindowsCE
  - RTLinux
    - Real time kernel runs Linux kernel as one of the real time processes
- Soft real time
  - ucLinux
  - Linux