NTNU | Norwegian University of Science and Technology

Operating Systems

Lecture overview and Q&A Session 7 – 28.2.2022

Michael Engel

Lectures 11 and 12

Inter-process communication

- Message-based communication
- Unix signals, pipes and message queues
- Sockets
- Remote procedure calls (RPC)

Uniprocessor scheduling

- Dispatch states: short/medium/long-term scheduling
- Scheduling algorithms
  - FCFS, RR, virtual RR, SPN, SRTF, HRRN
- Feedback scheduling and priorities
  - Multi-level scheduling
- Scheduling in Unix and Windows

IPC: Message-based communication

- Multiple processes can cooperate
- Communication using messages
  - exchanged (copied) between processes, no shared memory
  - operations: send and receive
  - synchronous ("rendezvous") or asynchronous
- Communication using shared memory
  - synchronisation is important
- Addressing: identifying communication partners
  - direct (e.g. process IDs) or indirect (e.g. pipes)
  - group addressing: multi/broadcast
- Also important
  - message format and transmission parameters

Unix signals, pipes and message queues

- Signals are *interrupts implemented in software*
  - minimal form of IPC: transmits signal number only
  - sent by `kill(2)` syscall or OS kernel, delivered asynch.
  - receiving process can choose to ignore *some* signals
  - or register a *signal handler* that called when signal arrives
- Pipes are *channels* between to communicating processes
  - unidirectional, buffered, reliable, stream oriented
  - operations: read and write (character order maintained)
  - blocks when pipe is full (write) and empty (read)
- Unix message pipes: “key” (unique) used for identification
  - undirected buffered M:N communication, typed msgs.
  - blocking & non blocking ops to send/receive messages

Sockets

- General communication endpoints in a *computer network*
  - bidirectional and buffered
- Abstract from details of the communication system
  - Socket "domains": Unix, Internet & more
  - Domains determine *protocol* that can be used (e.g. TCP)
  - Domains determine the *address family* (e.g. IP)
- Socket types:
  - stream vs. connection vs. message oriented
  - reliable vs. unreliable
- Socket operations:
  - socket, bind, sendto, recvfrom, listen, accept

Remote Procedure Calls

- RPC = *function call between different processes*
  - high grade of abstraction, usually in user mode library
- One RPC call maps to multiple messages
  - request: caller → callee
    - contains function name and parameters
  - response: callee → caller
    - contains result(s) or error message
- Used in many context
  - Network file system NFS (SunRPC)
  - Linux D-Bus

Dispatch states: short/medium/long-term scheduling

- logical state of process representing its dispatch state
  - short-term scheduling (state change µs – ms)
    - ready, running, blocked
  - medium-term scheduling (ms – minutes)
    - swapped and ready, swapped and blocked
  - long-term scheduling (minutes – hours)
    - created, terminated
    - using fork/exec/wait syscalls

Scheduling algorithms (1)

- First-Come First-Served – FCFS (non-preemptive)
  - first process arriving is executed first until it terminates
  - *convoi effect*: processes with short CPU bursts disadvantaged
- Round Robin – RR (preemptive)
  - processor time is split into time slices
  - when a time slice is used up, a process switch *can* occur
  - efficiency depends on chosen length of the time slice
- Virtual Round Robin – VRR (preemptive)
  - proc's are added to a preferred list when I/O burst ends
  - avoids unequal distribution of CPU times with RR
  - uses time slices of different lengths

Scheduling algorithms (2)

- Shortest process next – SPN (non-preemptive)
  - reduces disadvantage of short CPU bursts with FCFS
  - requires knowledge about the process run times
  - danger of *starvation* of CPU-intensive processes
- Shortest Remaining Time First – SRTF (SPN+preemption)
  - running process preempted when expected CPU burst or arriving process < remaining CPU burst of current one
  - *not* based on timer interrupts, nevertheless preemptive
  - processes can also starve using SRTF
- Highest Response Ratio Next – HRRN
  - considers *aging* of processes (waiting time R)
  - always selects the process with the highest value of R

Feedback scheduling and priorities

- Short processes obtain an advantage without having to estimate the relative lengths of processes (preemptive)
  - penalization of long running processes
- Multiple ready lists according to number of priority levels
  - priorities can decrease over time
- Short processes finish in a relatively short amount of time, but long processes can starve
- Priorities can be static or dynamic
  - static priorities are defined when a process is created
  - dynamic priorities are updated while a process is running

Multi-level scheduling

- Combine different scheduling strategies
- e.g. support of
  - interactive and background processing or
  - realtime and non-realtime processing
    - interactive / real-time critical processes are preferred
- implementation typically uses multiple ready lists
  - every ready list has its own scheduling strategy
  - lists are typically processed using priority, FCFS or RR
- Usually very complex

Scheduling in Unix and Windows

- Traditional Unix: two step preemptive approach
  - high-level: mid term, using swapping
  - low-level: short term preemptive, MLFB, dyn. proc. prio's
- 4.3 BSD Unix
  - Smoothing for processes that are woken up and were blocked for more than 1 second
- Windows NT
  - Preemptive, prio- & time slice-based *thread scheduling*
  - Priority classes
  - thread type determines time quantum available to the thread
  - adaptive priorities
    - *dynamic boost* temporarily increases prio of interactive processes

Overview Theoretical Exercise 5

All about scheduling

Why?

- Scheduling algorithms are central to the operation of a multitasking OS
- Many different approaches
  - No "golden solution" that is appropriate for all use cases
- Complex solution
  - enable better adaptation e.g. to dynamically changing requirements and operating conditions

Overview Practical Exercise 2

Communication and multithreading: **write a *multithreaded web server***

- Requires the combination of knowledge
  - IPC using sockets (lecture 11)
  - Multithreading (lecture 5)
  - Implementing advanced synchronization using counting semaphores and a variant of the producer/consumer problem (lecture 6/7)

- Available this evening: exercise sheet + ***header files*** with API definitions
  - Three weeks time, ***no extensions given! Start early!***
- We use *pthreads* as multithreading support library
  - Many tutorials on pthreads are available
  - This one is a short but useful one with code examples: https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html