Chapter 4-3: Processes and Operating Systems

Soo-Ik Chae

Topics

- Priority ceiling protocol (PCP)
- Performance estimation and Scheduling
- Languages and scheduling
- Operating systems mechanisms and overhead.
- Embedded file systems.

Priority ceiling protocol

PCP introduced by Sha, Rajkumar, Lehoczky 1990 as improvement of PIP

- Prevents formation of deadlock
- Prevents formation of chained blocking

Idea: extend PIP by a special granting rule for locking a free semaphore
- rule does not allow a job to enter a critical section if there are locked semaphores that could block it
- => once a job enters its first critical section it can never be blocked by lower-priority jobs.

Method: - assign a priority ceiling to each semaphore
- priority ceiling = priority of highest-priority job that can lock it
- job J is allowed to enter a critical section only if its priority > all priority ceilings of semaphore currently locked by jobs ≠ J

PCP Protocol Definition (1)

- Each semaphore S_k is assigned a priority ceiling $C(S_k)$ equal to the priority of the highest-priority job that can lock it. Note that $C(S_k)$ is a static value that can be computed off-line.
- Let J_i be the job with the highest priority among all jobs ready to run; thus, J_i is assigned the processor.
- Let S^* be the semaphore with the highest ceiling among all the semaphores currently locked by jobs other than J_i and let $C(S^*)$ be its ceiling.
- To enter a critical section guarded by a semaphore S_k, J_i must have a priority higher than $C(S^*)$. If $P_i \leq C(S^*)$, the lock on S_k is denied and J_i is said to be blocked on semaphore S^* by the job that holds the lock on S^*

PCP Protocol Definition (2)

- When a job J_i is blocked on a semaphore, it transmits its priority to the job, say J_k, that holds that semaphore. Hence, J_k resumes and executes the rest of its critical section with the priority of J_i. J_k is said to *inherit* the priority of J_i. In general, a task inherits the highest priority of the jobs blocked by it.

- When J_k exits a critical section, it unlocks the semaphore and the highest-priority job, if any, blocked on the semaphore is awakened. Moreover, the active priority of J_k is updated as follows: if no other jobs are blocked by J_k , p_k is set to the nominal priority P_k ; otherwise, it is set to the highest priority of the jobs blocked by J_k .

- Priority inheritance is transitive; that is, if a job J_3 blocks a job J_2, and J_2 blocks J_1 , then J_3 inherits the priority of J_1 via J_2 .

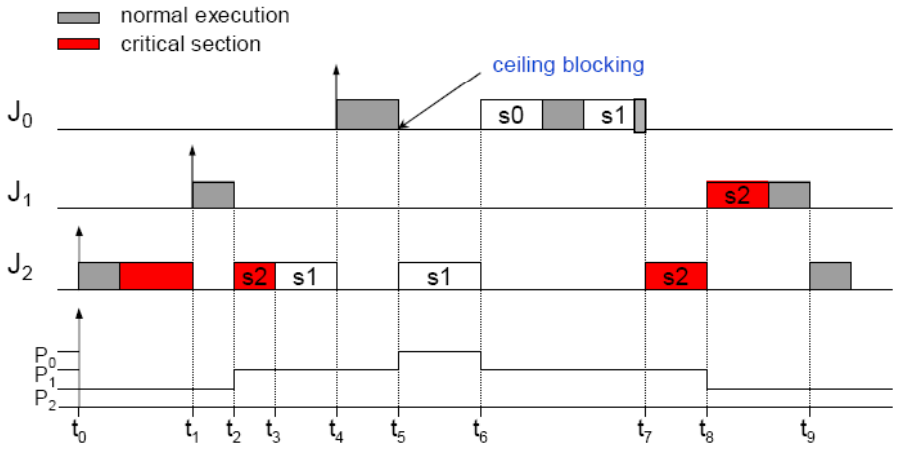
PCP Example (1)

Consider: 3 jobs J_0, J_1, J_2 with decreasing priorities
J_0 sequentially accesses critical sections
guarded by S_0, S_1
J_1 accesses only c.s. guarded by S_2
J_2 uses S_2 and than a nested-in
access to S_1

This results in the following priority ceiling of the semaphores:

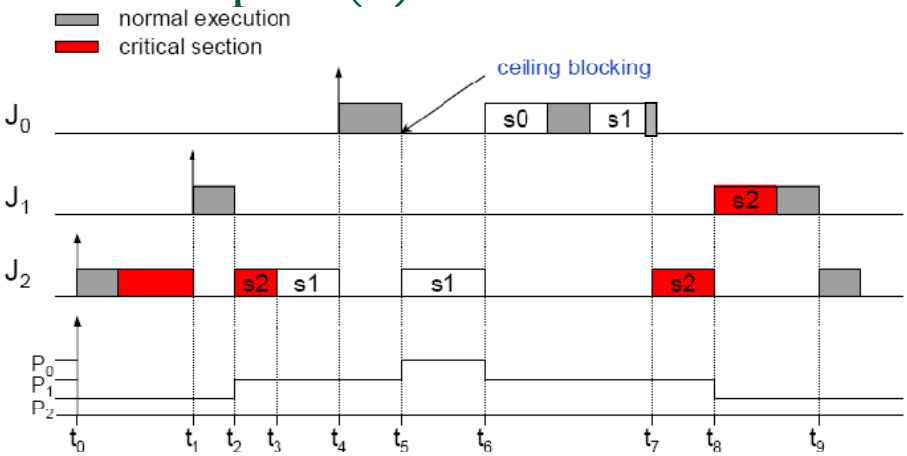
$C(S_0) = P_0$
$C(S_1) = P_0$
$C(S_2) = P_1$

PCP Example (2)



- At time t_0, J_2 is activated and, since it is the only job ready to run, it starts executing and later locks semaphore S_2 .
- At time t_1 , J_1 becomes ready and preempts J_2 .
- At time t_2 , J_1 attempts to lock S_2, but it is blocked by protocol because P_1 it is not greater than $C(S_2)$. Then, J_2 inherits the priority of J_1 and resumes is execution .
- At time t_3 , J_2 successfully enters its nested critical section by locking S_1 . Note that J_2 is allowed to lock S_1 because no semaphore are locked by other jobs.

PCP Example (3)



- At time t_4 ,while J_2 is executing at a priority $p_2 = P_1$, J_0 becomes ready and preempts J_2 because $P_0 > p_2$.
- At time t_5 , J_0 attempts to lock S_0 , which is not locked by any job. However,J_0 is blocked by the protocol because its priority is not higher than $C(S_1)$, which is the highest ceiling among all semaphores currently locked by the other jobs. Since S_1 is locked by J_2 , J_2 inherits the priority of J_0 and resumes its execution.

PCP Properties

Theorem: The priority Ceiling Protocol prevents deadlocks.

Theorem: (Sha-Rajkumar-Lehoczky) Under the Priority Ceiling Protocol, a job J_i can be blocked for at most duration of one critical section .

Performance Estimation (1)

- Assumption that the computation time of a process is fixed is not very realistic
- Execution time depends on
  - Data dependent path
  - Cache
- Strong interests on effects of multiple tasks on the cache
- A segmented locked cache
  - A program can lock a part of the cache so that no other program could modify those cache locations.
  - This would allow the program to certain parts of itself in the cache after a preemption.
  - It reduces the cache size not only for the program with the lock bust also for other programs in the system.

Performance Estimation

- Compiler support for software-based cache partitioning [Mueller]

> Cache memories have become an essential part of modern processors to bridge the increasing gap between fast processors and slower main memory. Until recently, cache memories were thought to impose unpredictable execution time behavior for hard real-time systems. But recent results show that the speedup of caches can be exploited without a significant sacrifice of predictability. These results were obtained under the assumption that real-time tasks be scheduled **non-preemptively**.

Software-based cache partitioning

> This paper introduces a method to maintain predictability of execution time within **preemptive**, cached real-time systems and discusses the impact on compilation support for such a system. Preemptive systems with caches are made predictable via software-based cache partitioning. With this approach, the cache is divided into distinct portions associated with a real-time task, such that a task may only use its portion. The compiler has to support instruction and data partitioning for each task. Instruction partitioning involves non-linear control-flow transformations, while data partitioning involves code transformations of data references. The impact on execution time of these transformations is also discussed.

Software-based cache partitioning

- Partitioning the cache for use by multiple processes.

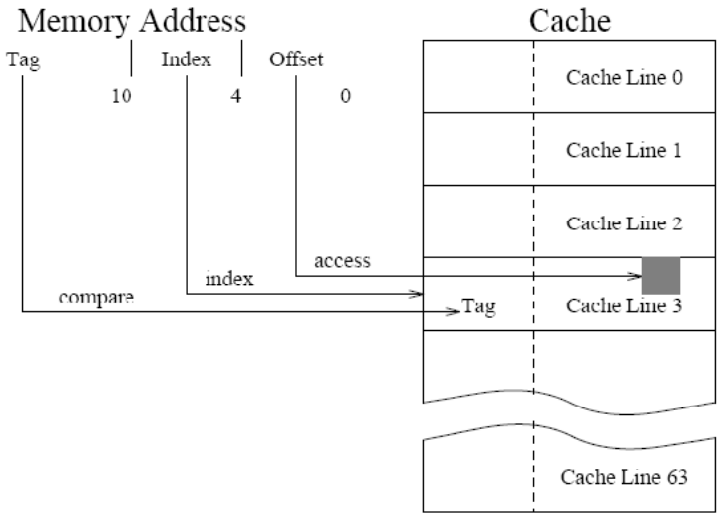


Figure 1: Indexing into a Direct-Mapped Cache

Software-based cache partitioning

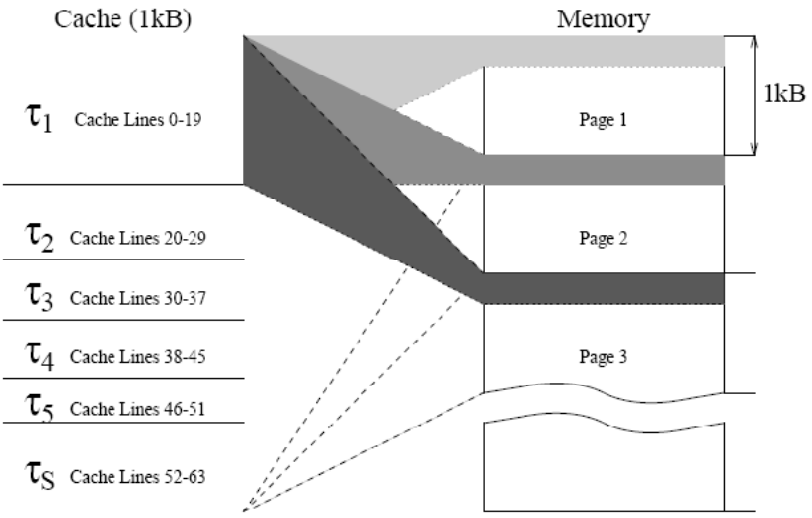


Figure 2: Cache and Memory Partitioning

Software-based cache partitioning

The code and the data of a task have to be restricted to only those memory portions that map into the cache lines assigned to the task. If the code/data size of a task exceeds its cache partition size, the code/data has to be scattered over the address space. In the above example, consider τ_1 with 10k instructions. The instruction space will be divided into 32 portions of 320B each, since τ_1 was given the first 20 lines (320B) in the cache. Thus, the first 320B within each 1kB page in main memory contain instructions of τ_1, up to page 32. (In the context of this paper, the memory page size is given by the cache size, which does not have to match the system page size.)

Software-based cache partitioning

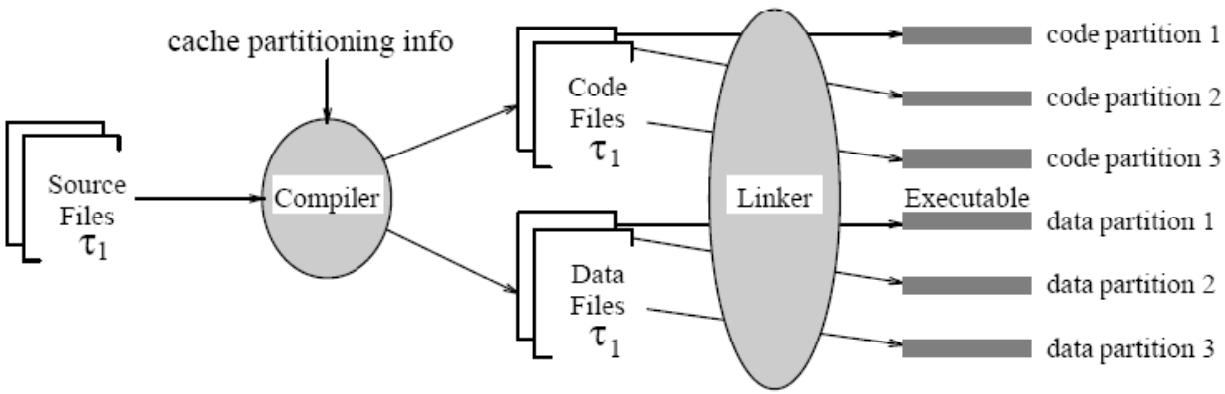


Figure 3: Compiling and Linking

Cache modeling and scheduling

- Li and Wolf developed a model for multitasking in caches.
  - each process has a stable footprint in the cache.
- Each process was modeled with a two-state model:
  - Process is in the cache.
  - Process is not in the cache.
  - Total state of the cache is given by the union of all the models for the processes that use the cache.
- The performance of a process is modeled by two major measured numbers
  - The worst-case execution time when the process is not in the cache
  - The average-case execution time when the process is in the cache.

Cache modeling and scheduling

- Given the cache model and the performance characteristics of each process, they constructed an abstract schedule that approximated the execution time of the multitasking system.
- Fig. 10: accuracy
  - Average error: less than 10%

Languages and scheduling

- Programming languages can capture information about tasks and inter-process communication.
- Compilation can generate specialized implementations, implement static schedules.
- Let look at languages that provide task-level models for system activity.

Codesign finite state machines (CFSMs)

- A control model explicitly designed to be implemented as combinations of hardware and software.
- A CFSM repeatedly executes a four-phase cycle:
  - Idle.
  - Detect input events.
  - Go to new state based on current state and inputs.
  - Emit outputs.
- A CFSM is modeled by an automaton with one-input buffers for each input and one-input buffers for each output.

Static scheduling using Petri nets

- Lin and Zhu developed an algorithm for statically scheduling processes using Petri nets.
- Given a Petri net model for a process, find maximal acyclic fragments of the process, schedule operations in each fragment.
- Expansion of a Petri net: acyclic Petri net in which
  - every transition has one input or output place,
  - at least one place with no input transitions, (initial places)
  - at least one place with no output transitions (cut-off places)

Static scheduling using Petri nets

- Maixmal expansion of G with respect to m, E is an acyclic Petri net with the following properties
  - The initial place correspond to m (initial marking)
  - The cut-off places correspond to the set of places encountered when a cycle has been reached
  - E is transitively closed: for each transition or place in E, all preceding places and transitions reachable form m are also in E
- Code is generated form maximally expanded fragment by pre-ordering operations.

Static scheduling using Petri nets

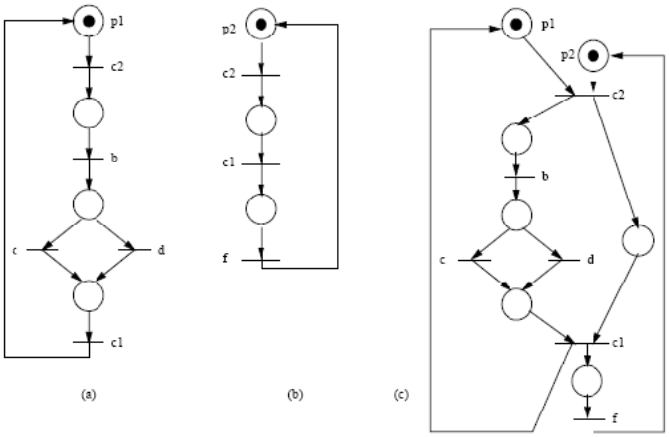
- Constructing a Petri net model from a program of communicating processes.

```
1 ping (input chan(int) a, output chan(int) b) {
2   int x;
3   for (;;) {
4       x = <-a; /* receive */
5       if(x <   0) x = 10 - x;
6       else x = 10 + x;
7       b <-= x; /* send */
8 } }

9 pong (input chan(int) c, output chan(int) d) {
10  int y, z = 0;
11  for (;;) {
12      d <-= 10;/* send */
13      y = <-c; /* receive */
14      z = (z + y) % 345;
15 }}

16 system ( ) {
17   chan(int) c1, c2;
18   par {
19       ping (c2, c1);
20       pong (c1, c2);
21 } }
```

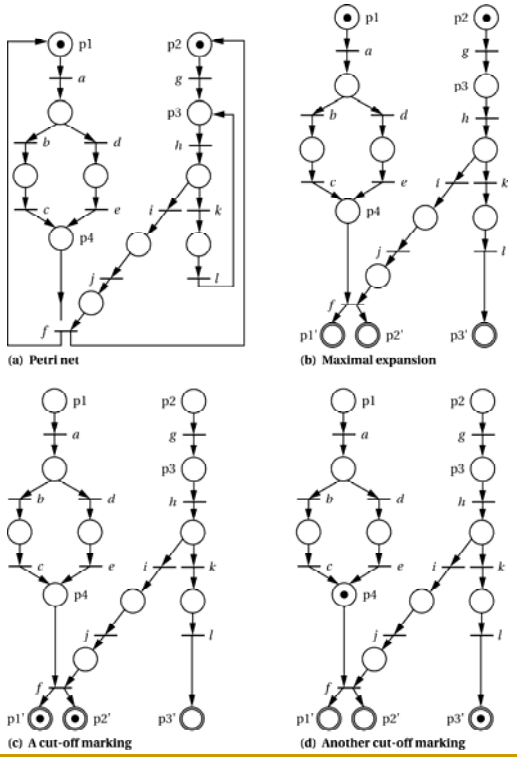

Static scheduling using Petri nets



(a)		(b)		(c)	
c2:	x=<-a	c2:	d<-=10	c2:	x=10
b:	(x<0)	c1:	y=<-c	b:	(x<0)
c:	x=10-x	f:	z=(z+y)%345	c:	x=10-x
d:	x=10+x			d:	x=10+x
c1:	b<-=x			c1:	y=x;
				f:	z=(z+y)%345

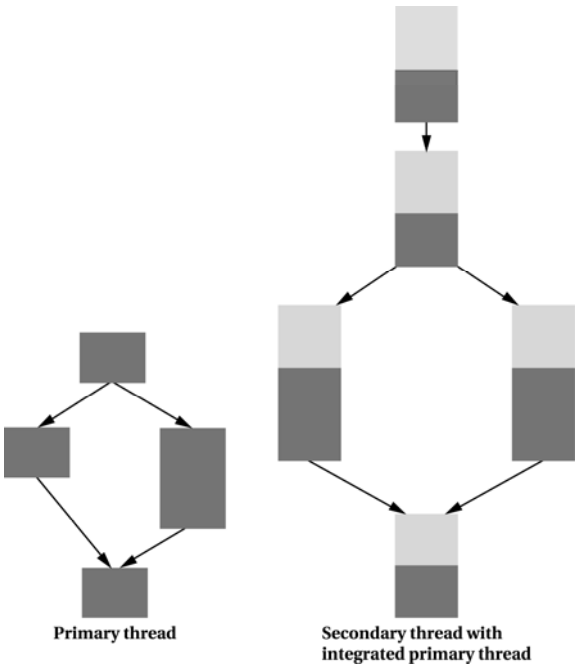
Fig. 1. (a) ping (b) pong (c) system = ping || pong

Maximal expansions



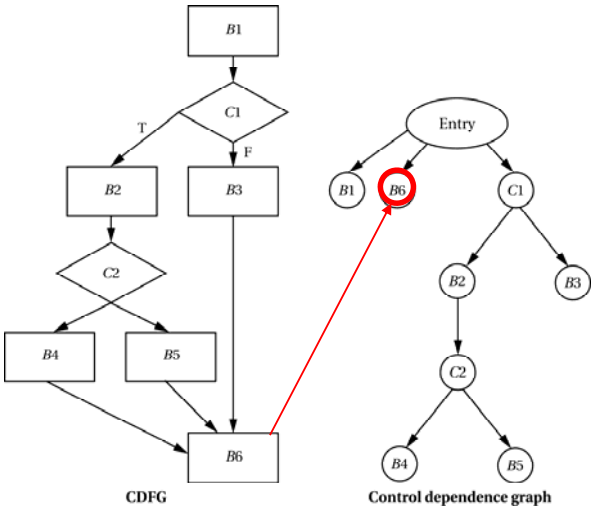
Software thread integration

- Dean: schedule multiple threads statically in a single program.
- Primary thread has real-time requirements.
- Secondary thread does not have real-time requirements.
- STI copies portions of the primary thread into the secondary thread so that the primary thread execute correctly and meets its deadlines.
  - No context switching
  - Reduce execution time



Software thread integration and CDGs

- Each thread is represented as control dependence graph (CDG).
  - Node: basic block/ conditional test
  - Edge: each control decision
  - Annotated with execution times of each blocks.
- Code from primary thread must be replicated in secondary thread to be sure that it is executed along every path.
  - Insertion points must meet primary thread deadlines.



General-purpose vs. real-time OS

- Schedulers have very different goals in real-time and general-purpose operating systems:
  - Real-time scheduler must meet deadlines.
  - General-purpose scheduler tries to distribute time equally among processes. (fairness and avoid starvation)
- Early real-time operating systems:
  - Hunter/Ready OS for microcontrollers was developed in early 1980s.
  - Mach ran on VAX, etc., provided real-time characteristics on large platforms.

Memory management

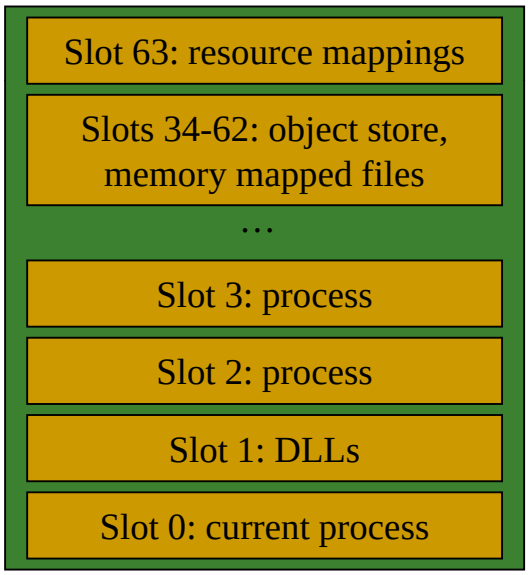
- Memory management allows RTOS to run outside applications. (protection)
  - Cell phones run downloaded, user-installed programs.
- Memory management helps the RTOS manage a large virtual address space.
- Flash may be used as a paging device.

Windows CE memory management

- Flat 32-bit address space.
- Top 2 GB for kernel.
  - Statically mapped.
- Bottom 2 GB for user processes.

WinCE user memory space

- 64 slots of 32 MB each.
- Slot 0 is currently running process.
- Slots 1-33 are the processes.
  - 32 processes max.
- Object store, memory mapped files, resource mappings.



Mechanisms for real time operation

- Two key mechanisms for real time:
  - Interrupt handler.
  - Scheduler.
  - Interrupt must be carefully handled to avoid destroying the real-time properties of the OS.
- Interrupt handler is part of the priority system. Interrupts have priorities set in hardware. These priorities supersede process priorities of the processes.
  - Can be seen as a distinct set of processes that are separate form the operating system's regular processes.
  - Should spend as little time as possible in the interrupt handlers that are dispatched by the hardware interrupt system.

Interrupt handling in RTOSs

- But many real devices need a significant amount of computation to be done somewhere.
- As a result, device oriented processing is often divided into two sections: an interrupt service routine (ISR) and an interrupt service thread (IST).
- ISR is dispatched by the hardware interrupt system while IST is a user-mode process
- Scheduler determines ability to meet deadlines.

Windows CE interrupts

- Two types of ISRs:
  - Static ISRs are built into kernel, one-way communication to IST.
  - Installable ISR can be dynamically loaded, uses shared memory to communicate with IST.

Static ISR

- Built into the kernel.
  - SHx and MIPS must be written in assembler, limited register availability.
- One-way communication from ISR to IST.
  - Can share a buffer but location must be predefined.
- Nested ISR support based on CPU.
- Stack is provided by the kernel.

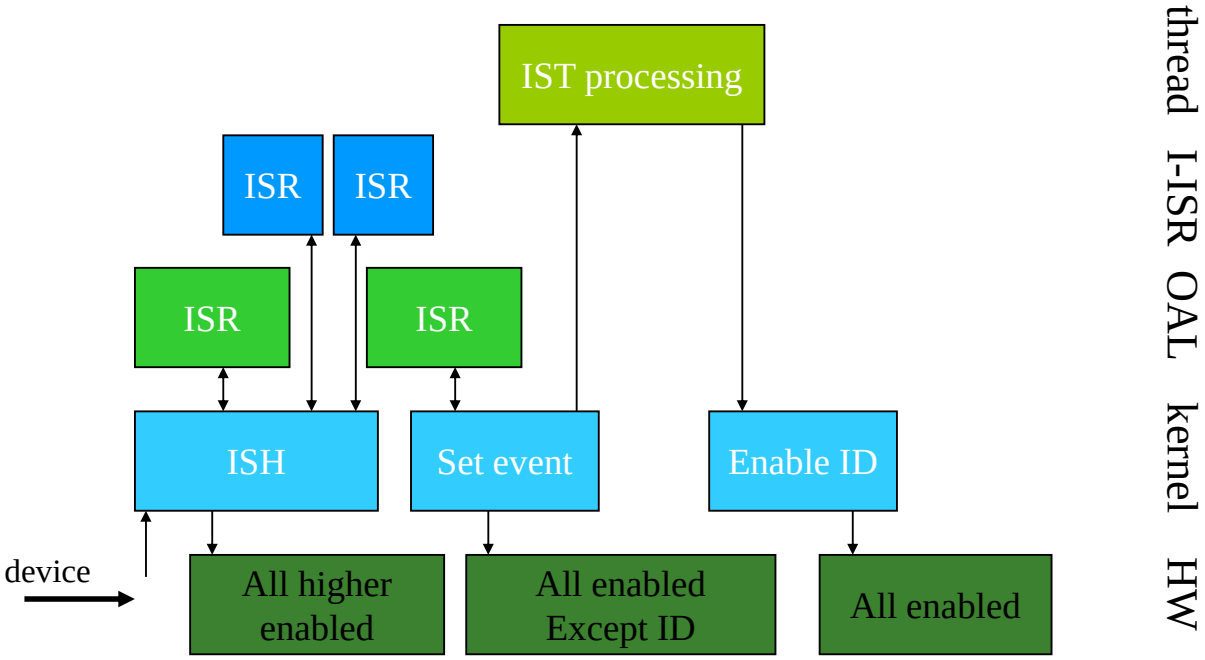
Installable ISR

- Can be dynamically loaded into kernel.
- Loads a C DLL (dynamically linked library).
- Can use shared memory for communication.
- ISRs are processed in the order they were installed.
- Limited stack size.

Interrupt latency

- ISR
  - Amount of time that interrupts are turned off
  - Time required to visit ISR, save registers, etc.
- IST
  - Time spent in ISR
  - Time spent in Kernel call
  - Thread scheduling time

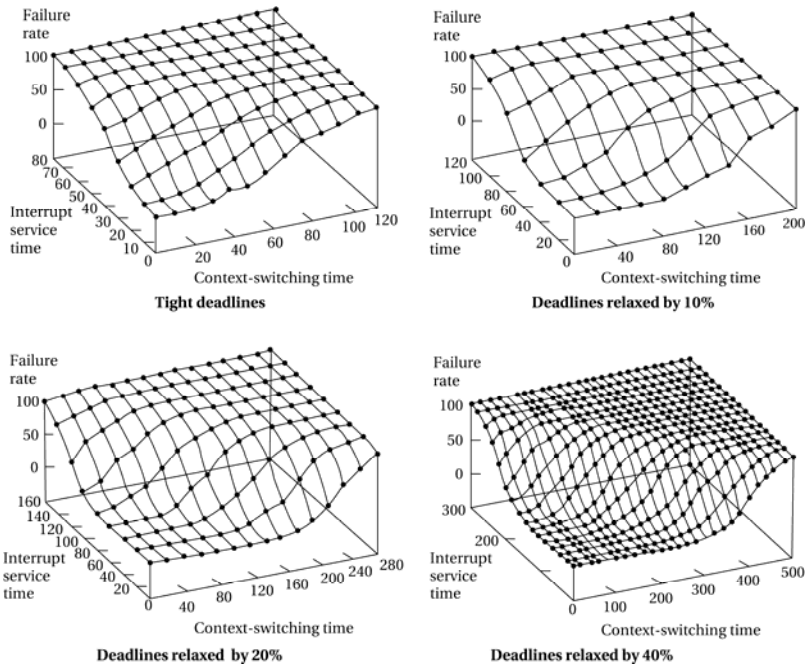
WinCE 4.x interrupts



Operating system overhead

- Rhodes and Wolf studied context switching overhead using simulation.
- Two-CPU system with bus.
- 100 random task graphs.
- Varying amounts of slack: none, 10%, 20%, 40%.

OS overhead results



[Rho99] © 1999 IEEE Computer Society

Interprocess communication

- IPC often used for large-scale communication in general-purpose systems.
- Mailboxes are specialized memories, used for small, fast transfers.
  - A writer and many readers
- Multimedia systems can be supported by quality-of-service (QoS) oriented inter-process communication services.
  - Streaming and large transfers

Power management

- Advanced Configuration and Power Management (ACPI) standard defines power management levels:
  - G3 mechanical off.
  - G2 soft off: OS need to be rebooted when leaving the state
  - G1 sleeping.
  - G0 working.
  - Legacy state: NON-ACPI power modes.

Embedded file systems

- Generally means flash memory storage.
- Many embedded file systems need to be compatible with PCs.
- Some file systems are primarily for reading, others for reading and writing.

Flash memory characteristics

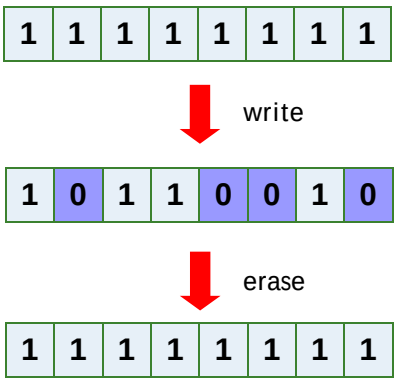
- Flash is electrically erasable.
- Two types of flash:
  - NOR flash operates similar to RAM.
    - Used to store executable code
    - More reliable, faster read, random access capability
  - NAND is block oriented, gives more transient failures.
    - Higher density, lower cost, faster write and erase time
    - Longer re-write life expectancy
    - Prone to single bit errors
  - NAND gets faster, may dominate in future.

Flash memory characteristics

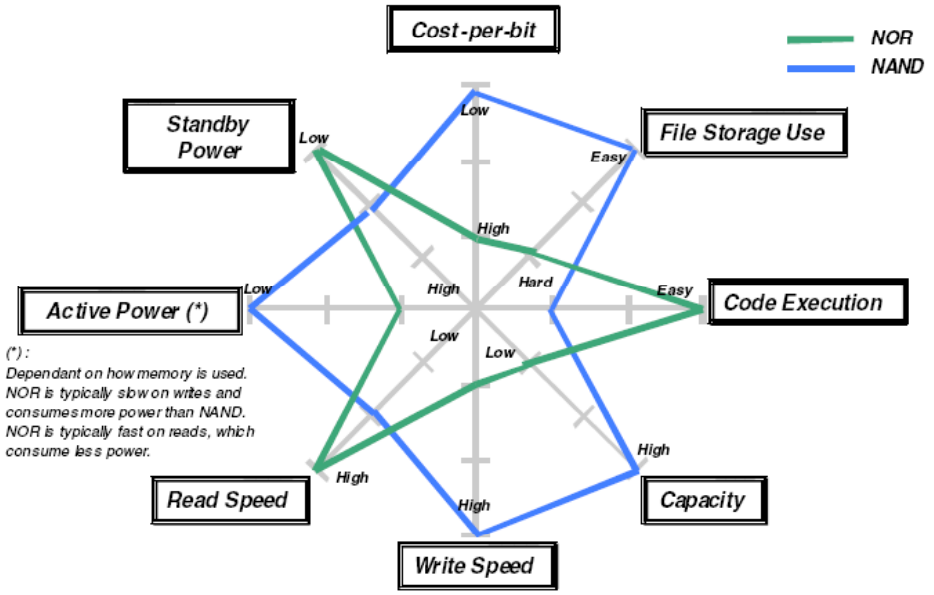
- Block writing
  - Flash cannot be written word-by-word as with RAM.
  - Flash memory must first be erased in large blocks and then written
  - Block size may be as large as 64KB
  - Erasing a block is considerably larger than a typical magnetic disc sector.
- Flash memory wears out during writing.
  - Early memories lasted for 10,000 cycles.
  - Modern memories last for 1 million cycles.

Flash Memory Characteristics

- Operations
  - Read
  - Write or Program
    - Changes state from 1 to 0
  - Erase
    - Changes state from 0 to 1
- Unit
  - Page (sector)
    - Read/Write unit (in NAND)
  - Block
    - Erase unit



Comparison of NOR and NAND Flash



Comparison of NOR and NAND Flash

	SLC NAND Flash (x8)	MLC NAND Flash (x8)	MLC NOR Flash (x16)
Density	512 Mbits[1] – 4 Gbits[2]	1Gbit to 16Gbit	16Mbit to 1Gbit
Read Speed	24 MB/s[3]	18.6 MB/s	103MB/s
Write Speed	8.0 MB/s	2.4 MB/s	0.47 MB/s
Erase Time	2.0 mSec	2.0mSec	900mSec
Interface	I/O – indirect access	I/O – indirect access	Random access
Application	Program/Data mass storage	Program/Data mass storage	eXecuteInPlace

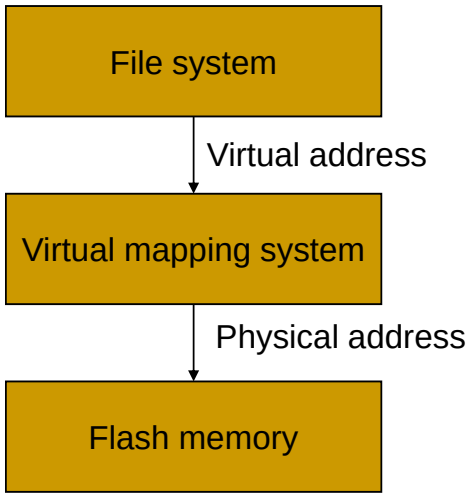
Figure 4: NAND and NOR Flash Operating Specifications

Wear leveling

- Flash memory systems move data to equalize wear during writes.
- File allocation table gets the most writes---must be moved as well.
- Formatting avoids multiple writes to file allocation table.

Virtual mapping

- Virtual mapping system stands between file API and physical file system:
  - Schedules erasures.
  - Consolidates data to empty an entire block
  - Identifies bad blocks.
  - Moves data for wear leveling.
- Virtual mapping system keeps a table to translate virtual to physical addresses.

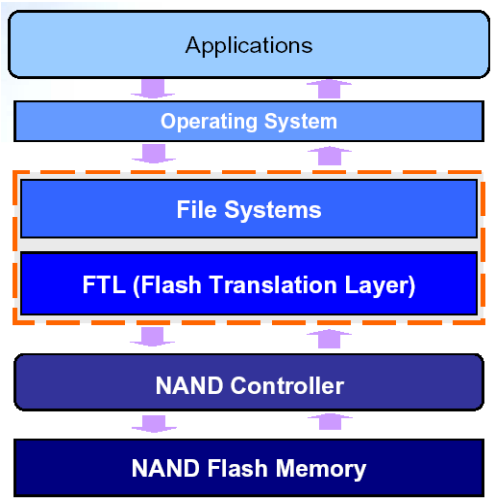


Log-structured file system

- Stores log of changes to file, not the original file.
  - Also known as journaling.
  - Developed for general-purpose systems, useful for flash.
- Journaling Flash File System (JFFS) maintains consistency during power losses.
- Yet Another Flash Filing System (YAFFS) is log-structured file system for NAND flash.

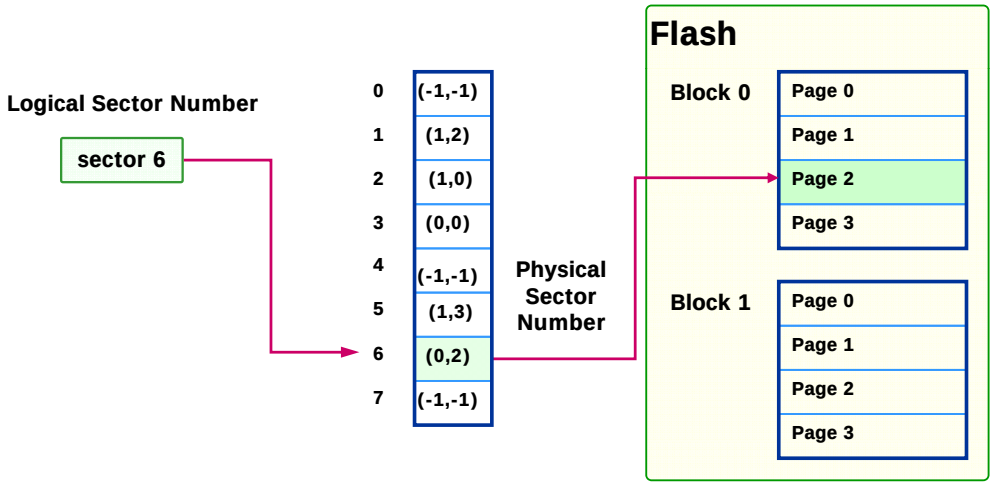
Flash Translation Layer (FTL)

- A software layer emulating standard block device interface
  - Read/Write
- Features
  - Sector mapping
  - Garbage collection
  - Power-off recovery
  - Bad block management
  - Wear-leveling
  - Error correction code (ECC)



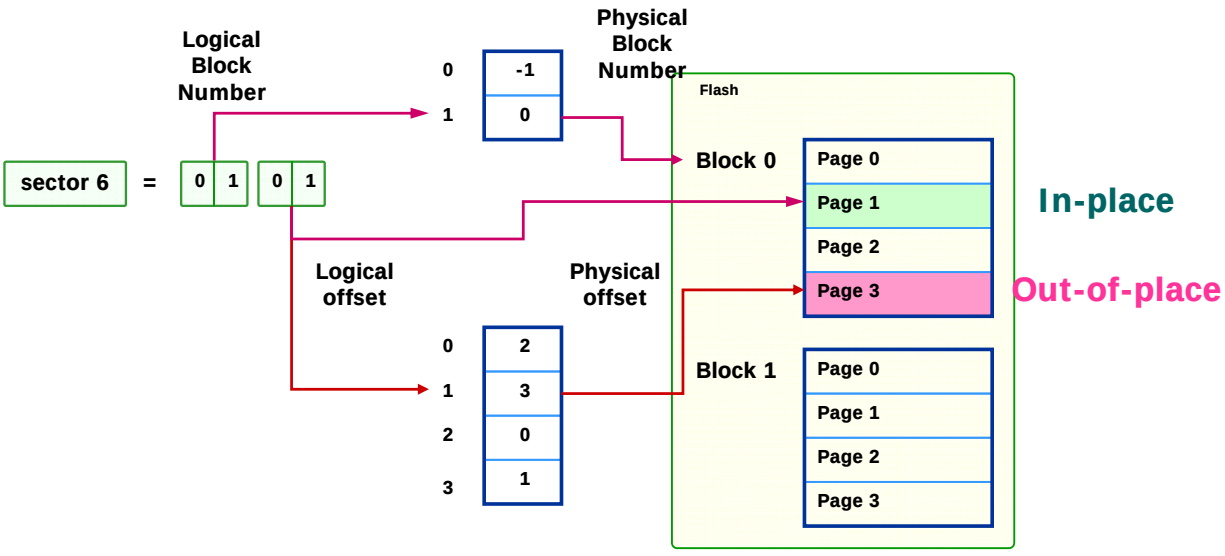
FTL Mapping Scheme (1)

- Page mapping



FTL Mapping Scheme (2)

- Block mapping



Flash-Aware File System

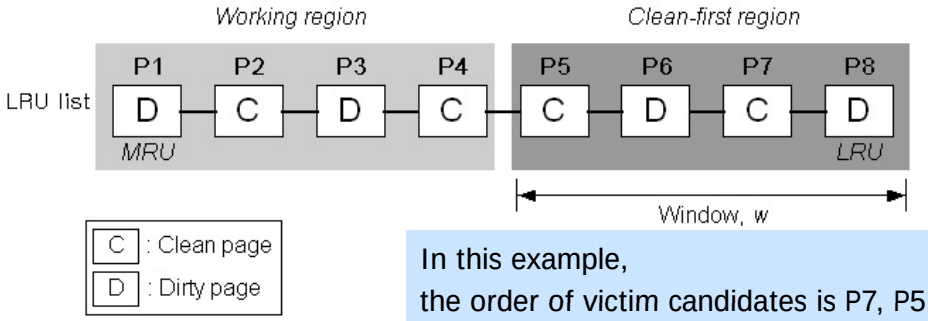
- File systems manage raw flash memory directly.
- More opportunities to optimize the performance.
- File system comprises in some FTL functionalities.
  - Sector mapping, garbage collection, wear-leveling, power-off recovery, etc.
- Example:
  - JFFS, JFFS2, YAFFS
- Limitation
  - Need to change the host operating system or special device driver to access flash-aware file system

Flash-Aware Demand Paging (1)

- Problems with flash memory
  - Write cost is much higher than read.
    - Time and energy consumption of a write operation is about six times higher than a read operation in NAND flash.
    - Write operations accompany potential erase operations.
  - Lifetime of flash memory is limited by write/erase.
- Replacement policy for flash memory
  - Should reduce the number of write operations on flash memory

Replacement Policy: Clean first LRU

- Dividing LRU list into two regions
  - Working region is suspected to have high cache hit rate
  - Clean-first region preserve dirty pages to reduce flash write operations
- First, it evicts clean pages in clean-first region.
- If it does not satisfy, evicts dirty pages.



In this example,
the order of victim candidates is P7, P5, P8, and P6.