TTK4147 - Notes

Jon Eirik Lisle Andresen

December 2017

Contents

1 Hardware/CPU

The hardware which is important in this course is a basic understanding of what a CPU is and what it does. A CPU is basically a set of registers, and ALU (Arithmetic Logic Unit), a set of instructions and IO.

1.1 Micro instructions

Any program that is executed on a computer is a set of smaller instructions. A instruction execution can be divided into two steps: 1. Load the instruction from main memory (RAM) into the instruction register according to the program counter (PC a register in the CPU that points to where the next instruction is found) and 2. execute the instruction. By default the PC is incremented after each execution. Then if the program says so fetch the next instruction.

The data in the IR corresponds to a micro instruction which the CPU can do. It's instructions is implemented as a large set of registers setting up the different buses, registers, IO and the ALU.

Categorize instructions into four categories:

1. Processor - memory. For processor memory communication.
2. Processor - IO. For processor IO communication. E.g disk or screen.
3. Data processing. Manipulating data.
4. Control. If, branch other change in sequence of execution.

1.2 Interrupts

As the name suggests it is a signal that interrupts what the CPU is doing and forces it to run some other predetermined code. This can be used as a way of responding to non deterministic events such as a button push without resorting to polling.

Categorize interrupt sources into four categories:

1. Program. Just as a result of some program execution
2. Timer. A set time causes an interrupt to happen.
3. IO. An IO signal goes to the interrupt pin on the CPU

4. HW - failure.

After each execution is complete the processor checks it an interrupt has happened. If it has then execute the interrupt handler routine. When this is done the CPU will save its state some where so that it can continue where it was interrupted once done. Else, disable interrupts and fetch the next execution to be run. Will generate some context switch and thus overhead, but only when an interrupt happens and the positives far outweigh the negatives.

If multiple interrupts happen there are two ways of dealing with it.

1. Disable interrupts while an interrupt is handled.
2. Prioritize such that a higher priority interrupt can interrupt a lower one, but not the other way around.

1.3 Memory and memory hierarchy

Often useful to implement several layers of memory with different speeds and capacities. The high speed, but low capacity (Cache) being close to the CPU and the low speed high capacity (RAM) being "far away". The more frequently used the data, the closer to the CPU it will be saved. The cache is not visible to the OS.

The main memory is too slow, so a bunch of instructions is loaded into the cache at a time. If you need one instruction there is a good chance you'll need the next 10 as well (Principle of locality).

1.3.1 Cache implementation

When writing a block to the cache another block has to be removed. Most often the one that hasn't been referenced in the longest time will be replaced (Least recently used (LRU) algorithm).

The data in cache has to be written back to main memory. When to do so is determined by write policy. Compromise between many write operations and having an obsolete main memory state.

1.4 IO operations

1. Programmed IO - The CPU polls device for new data and is busy while doing so.

2. Interrupt driven IO - Request data and receive an interrupt when it can receive the data an place the data in memory.
3. Direct Memory Access (DMA) - Other device responsible for IO. CPU sends request and receives interrupt from the DMA module when the DMA module has loaded the IO - data into memory. Best time i.e. least CPU usage, but additional circuit required.

2 Operating System(s)

The operating system is a layer between the user/developer and the hardware. Removes need to develop HW - specific programs.

Three ways for programs to interface with the OS and HW.

1. Instruction Set Architecture (ISA) - The way the OS talks to the HW. Defined by the CPU and saved in registers (to the right). Sometimes there are libraries that allow programs to access the ISA without using the OS.
2. Application Binary Interface (IBA) - Libraries which are a part of the OS containing functions and services that programs use to interface the ISA.
3. Application Programming Interface (API) - Same-ish as ABI's, but can be outside the OS.

The operations system also includes a scheduler allowing for more programs / threads to run in series making it look like more programs are running at the same time even when there is only one CPU core. The OS also controls the memory management such that programs just get the memory they need.

The OS has to keep track of resources and other memory and which process has access. Also keep track of files.

2.1 Kernel

Two main different types of kernel. Either **monolithic** or **micro** kernel. A monolithic kernel contains all the OS code in one large block where all programs and services have access to ALL the things.

A micro kernel only contains what is needed for kernel mode operation:

1. Scheduling

2. Memory management
3. IPC

2.2 FreeRTOS

Very lightweight OS. Consists of three main code files plus architecture specific code. Usable on micro-controller. Is free, but must document usage.

2.2.1 Tasks/Processes

Task is the same as a process. Have their own stacks. Has the following states:

Ready Ready to run, but no able to since a task with priority is in the running slot.

Running on the CPU

Suspended Put here by calling vTaskSuspend(). Returned by xTaskResume()

Blocked By waiting on a semaphore, timing or other things

If there are severe RAM constraints one can use **Co-Routines**. They have the same states except suspend and the all share a singe stack.

When the scheduler is startet it creates and idle task. The job of the idle task is to free up memory whenever other tasks have completed. This will only run when the CPU is idle.

2.2.2 Scheduling

Two modes. Either co operative or preemptive. In co-operative a task will run until it is blocked or yields control.

The scheduler is activated at a timer interval and if preemptive it will check for any higher priority task to run. Equal priority tasks are scheduled with Round Robin. 5.2.3

2.2.3 Inter-process communication

Has the normal mutexes, semaphores and queues (message passing). Also has its own **task notification** which uses less RAM.

2.3 Linux

Free and open source. May different variants called distros. There are many libraries with functionality that are a part of Linux. A static library is compiled with a program, a shared library is pre-compiled and can be shared between programs.

2.3.1 Kernel

Linux uses a modular monolithic kernel, meaning modules can be added to the kernel. These modules are to be relatively independent. These can be linked and unlinked to the kernel while the kernel i running.

2.3.2 Concurrency

Four mechanisms that are relevant today:

Atomic Operations Things that seem indivisible by the outside.

Spinlocks Mutexes where the waiting task spins instead of sleeping.

Semaphore Semaphores for kernel-space code.

Barriers Method for enforcing order of execution.

2.3.3 Memory Management

Linux uses a paging system to manage memory. The virtual address contains:

1. Global directory
2. Middle Directory (Set to 1 in 32 bit Linux)
3. Page Table
4. Offset

These together points to a place in physical memory.

Pages in memory will be replaced according to a ”least frequently used policy”. Each page has a counter that is incremented each time it is accessed and Linux will decrease each one every once in a while. If a counter reaches 0 that page will be a candidate for replacement.

The kernel allocates and deallocates memory according to a buddy system.

2.3.4 Scheduling

Three classes of scheduling used by Linux:

1. SHCED_FIFO Fifo RT threads
2. SHCED_RR Round robin RT threads
3. SHCED_OTHER Other non RT threads

The kernel schedule kernel threads individually regardless of whether they are threads or processes in the user space.

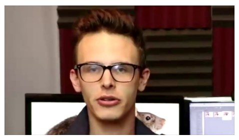
This is OK on paper, but the kernel is not really preemptive. There are also many "Linux-things" running in the background which can take an unknown amount of time.

2.3.5 Threads

Both processes and threads are running as separate kernel threads. Threads in a process share virtual memory. Faster context switches between threads than processes. Most threads are implemented in the user space.

2.3.5.1 Daemons

Programs that run in the background and does not interact with the user. Started by a program calling fork() which starts a child and then kills itself.



2.3.6 Kernel Space

Develop here when you need access to things not supported by some library in the user space.

Note that root $\neq$ kernel space. Root is highest privilege possible in user space.

2.3.7 Make Linux Great (RT) again

One can make Linux more RT by reducing the size of the OS and can make the scheduler more preemptive.

This is not great. It's OK, but not great. Alternatives are:

PREEMPT_RT Make less of the kernel unpreemtable by replacing spinlocks with mutexes, effect uncertain. There are parts of the Kernel that cannot be preempted.

Co-kernel Small kernel running below adding another layer for threads to run on.

Outsource Use a micro controller for the hard RT tasks.

2.3.8 Embedded Linux

Free, runs on everything and has a large community, but no real RT. Hard

2.4 QNX

Micro kernel, POSIX compliant. Great for RT and embedded applications. However it is proprietary.

2.4.1 Kernel

Here are some of the services the QNX kernel provides

Thread Services

Signaling

Message passing

Synchronization

Timer

Process management

A microkernel is in general good for RT.

2.4.2 Scheduling

Highest priority will run. There is inheritance. Uses a sporadic server 5.4.1. The SS is implemented as a thread running with a normal priority whenever it has budget left. When budget is empty, priority drop.

A thread is what the scheduler schedules. A process is a container for threads.

There are 256 priorities in QNX. 0 is idle. 1-63 is user configurable. 64-255 is for root only.

The scheduler will run when one of these happen:

1. Thread is blocked
2. A higher priority thread is ready to go.
3. The running threads yields control.

Uses nested interrupts and interrupts are almost always enabled.

2.4.3 Processes and threads

Building blocks for the system. They are visible to each other and can communicate. Threads are invisible to the outside.

Threads in a process share a heap and instructions. All threads have their own stack.

2.4.4 Message passing

One process creates a channel and another connects to it. A low priority task will inherit priority of a high one sending it a message wanting reply. Same applies the other way around. Can send a **pulse** which is a message without waiting for a reply.

2.5 VxWorks

Micro kernel. Used everywhere. Scaleable. Small!

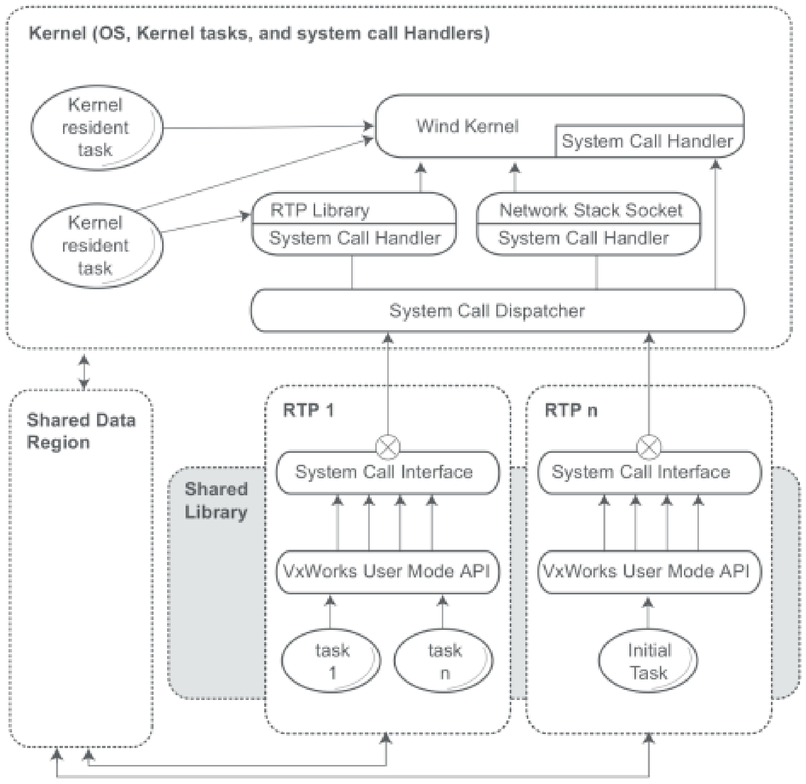
2.5.1 Memory Management

Used to skip it, but newer version has it.

All virtual addresses in the system are unique.

2.5.2 Task/Process/Thread

Task is the same as a process.



Tasks can be in more than one state.

3 Processes

A process is a thing that the process can run. Consists of

1. Program code - Whatever the program is designed to do
2. Metadata - Name, process id (PID), priority, memory pointers, sate and other things.

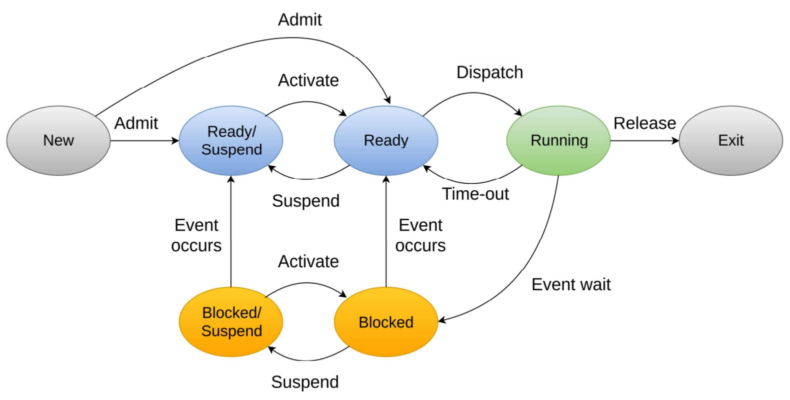
Process image is something the CPU/OS takes of the process when memory is being allocated to it, but before it starts execution. Contains:

1. User data - Modifiable part of user space. Contains program data, user stack and other programs.
2. User program - The program to be run.
3. A stack - each process has at least one LIFO queue (stack) that belongs to it. Here parameters and pointers for other procedures are stored.
4. Process Control Block - Data needed by the OS to control the process.

3.1 Seven-state Process Model

Way to describe different states a process can take.

Seven-state Process Model



New Processes that are created, but not yet added to the pool of runable things.

Ready/Suspend Processes that are ready to run, but are suspended.

Ready Processes that are ready!

Running What the CPU is currently working on.

Blocked Queue for processes that are waiting for an event to occur.

Blocked/Suspend Processes that are blocked AND suspended.

Exit Processes that are done with running, but have yet to be deleted from memory.

A suspended process has been swapped i.e. moved to disk. The disk is much slower than memory, but has ”infinite” capacity.

Processes can run in at least two modes depending on the OS. This is to protect the OS from harm.

User mode What programs created by the user runs in. Cannot do everything.

System mode AKA kernel mode. Has access to ALL the things.

A process is created by:

PID OS assigns a process identifier.

Memory OS allocates a large enough chunk of memory to the process.

Control block Init the control block of the process.

Add to list OS puts the process in the appropriate list in the SSPM.

Other tasks If there are any OS specific ones.

The switch between which process is running are as follows:

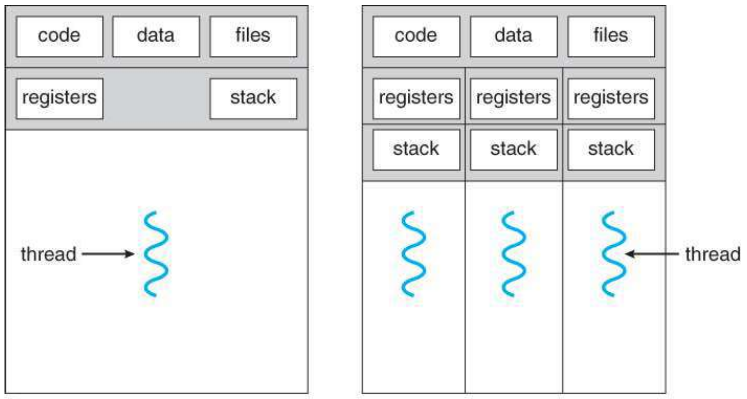
1. Save the context of the processor.
2. Update control block of currently running process.
3. Move the control block of running process to the right queue.
4. Find some other process to run.
5. Update that processes control block.
6. Update memory management data structures.
7. Restore the state the CPU had at the time the now ready to run process was last switched out.

3.2 Threads

A thread is the part of the process that actually runs. A process can have more than one thread when running on an OS that supports multithreading.

The other part of the process is the part that owns a resource and data. This is then shared among the threads. All threads of a process share the same address space.

Each thread has its own registers and stack.



Threads are faster to create and switch between. Since they share some resources it is easier to synchronize them and get them to communicate, but at the cost of less protection.

As with processes we have both user level and kernel level threads. For user level the kernel does not know about the threads and the scheduling is done by the application itself. Is quick and easy. For kernel level the kernel can schedule another thread if a thread is blocked. Also enables different threads from the same process to run on different CPU-cores. A combined approach is possible where user level threads are mapped to kernel level ones.

3.3 Inter process communication

Processes and threads are to communicate to share info and to synchronize. Three constructs for synchronization in this course: semaphores, mutexes and message passing.

A race condition is when an error emerges as a result of unfortunate order of execution.

3.3.1 Semaphores

Semaphores have an integer value and has three operations that can be done to it:

Initialize Set the value of the semaphore to a non negative value.

Wait Decrement the value. If the result is negative the caller will be blocked.

Signal Increment the value. If the result is ≤ 0.

A semaphore can be counting i.e have any value up to the initialized value or binary.

3.3.2 Mutex

A mutex is a binary semaphore WITH ownership. They are used to protect shared resources.

A mutex can be initialized, locked and unlocked. A mutex can be recursive i.e. a process can lock it multiple times. The mutex then has to be unlocked an equal amount of times to be taken by someone else.

3.3.3 Message passing

Do you even Golang?

4 Memory Management

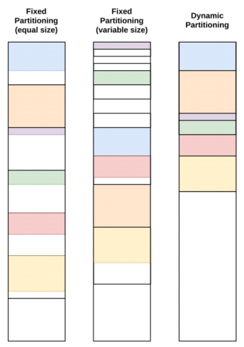
Easy when you have just one program and OS. OS gets what it needs and the program gets the rest. But this cannot be known when there are more programs running. Rarely known how much a program will need when started. Also have to be able to swap a program to disk and put it back in any location.

Memory management system have to take protection into account: One process should not be able to access an other process' memory, no one can change the kernels memory and these protections mechanisms should be implemented in HW on the CPU for speed. However processes need to cooperate so the protection has to allow for communication.

4.1 Memory Management Techniques

The size of the chunks of memory can be implemented in different ways.

When chunks of memory is occupied and the unoccupied there can be fragmentation which is bad. We mention three ways of placing memory.


Fixed Partitioning (equal size)
Fixed Partitioning (variable size)
Dynamic Partitioning

Best-fit Choose the block that is closest in size to the requested chunk size.

Fist-fit Begin scanning from the beginning of the memory and take the first that is large enough.

Next-fit Begin scan from the place where memory was last placed and take the first that is large enough.

4.1.1 Buddy system

Memory is not in fixed sizes. There is some smallest possible partition size = 2^L for some integer L and a largest = 2^B for some other integer B. The buddy system will split memory into two and then split that in two until some appropriate size is found. Is OK, but will cause internal fragmentation.

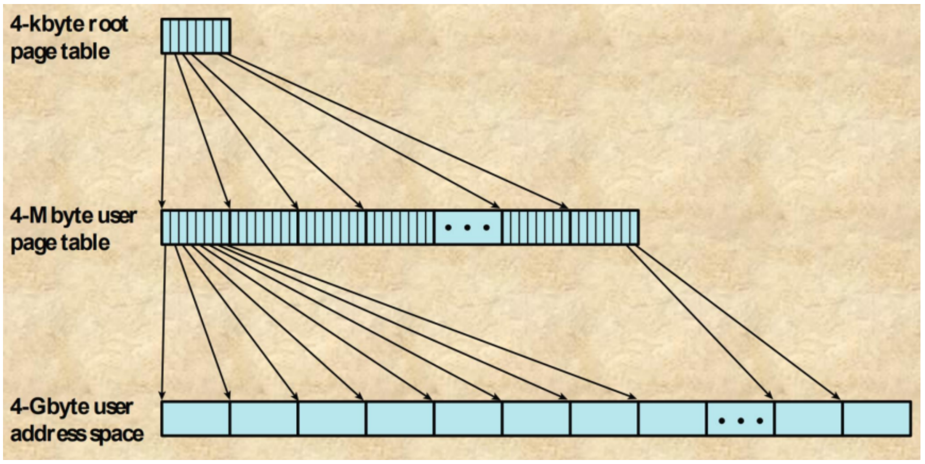
4.2 Paging and segmentation and logical addresses

A logical address is an address with a starting point and a relative address from there i.e go to page nr. 3 and go 154 steps from there.

Both divide memory of a program into parts and keep track of where it is. Paging create fixed size chunks. Segmentation create variable length chucks or words. Both can be used in logical addresses.

4.2.1 Multi-level paging

Each address in a page points to another paging table which again can point to another table. These first page look-ups will be your starting points in the subsequent use of logical addresses.



It is possible to combine paging and segmentation in a multi-level scheme. Will look like segmentation to the user and programs and paging to the system.

4.3 Virtual memory (VM)

Add the disk as a part of the main memory. A process can then have some of its data in disk and some in ram. Advantages: Can use memory only for data being used at the time. The disk is ”unlimited”. Also provides more memory isolation protecting processes from accessing memory that isn't theirs. Disadvantages: Thrashing i.e the system spending most of its time moving things between memory and disk.

VM is not great for RT as it is unpredictable.

VM is often implemented on a separate chip on the CPU called a memory management unit (MMU). On the MMU there is also a translation lookaside buffer (TLB) which is a buffer saving the translations between VM and physical memory most recently used.

5 Scheduling

Scheduling is a strategy for deciding which proccess is to run over time. Can be divided into:

Short-term Which of the ready processes to run. Aka dispatcher.

Medium-term Which processes that are to be swapped in or out.

Long-term Which processes are to be added to the ready queue.

There are several criteria one can choose when making a scheduler.

Criteria for scheduling seen from the user:

Turnaround time Time from adding a task until it is completed.

Response time Time from adding a task until it is started.

Deadlines When a process has to be completed.

Predictability A given job should run at same time and cost regardless of system load.

Criteria for scheduling seen from the system:

Throughput Maximize processes completed pr time.

Utilization	Minimize idle time of the CPU.
Fairness	Processes should be treated the same and no one should starve.
Enforcing Priorities	A higher priority should always be able to run.
Balancing resources	If a resource is overused scheduler should run a process that does not require that resource.

5.1 Dispatcher

The dispatcher will be called when: A timer sets it of, I/O changes something, an error exits the current process or a process causes itself to be blocked or wait.

5.2 Scheduling/dispatcher schemes

5.2.1 Priority based scheduling

Assign priorities to processes either statically or dynamical. Each priority level gets it own queue and the scheduler starts at the top an tries to keep them all empty. Lower priority might suffer from starvation.

5.2.2 First-come-first-served

Dispatcher chooses the one that has been waiting the longest. Favors long running time tasks. Favor CPU use over IO usage.

5.2.3 Round robin

Regular timer interrupts. The running process is then placed in the back of a FIFO queue with the rest of the ready to run queues. Will favor short processes.

5.2.4 Shortest Process Next (SPN

Dispatcher chooses the process with the least execution time. Requires knowledge of running time. Long processes might starve and cannot really have infinite tasks.

5.2.5 Shortest Remaining Time

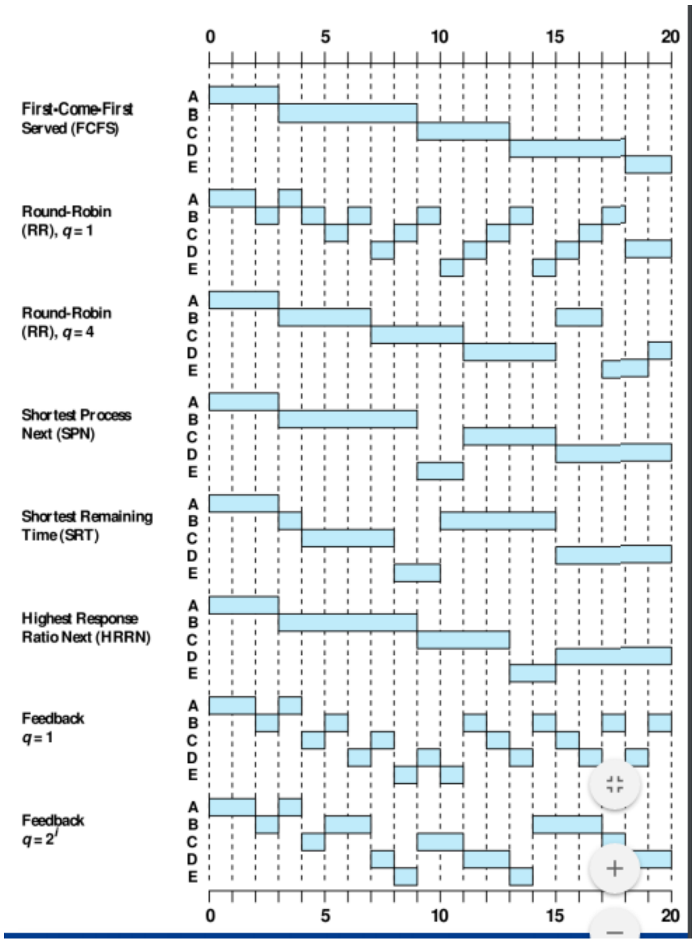
Preemptive version of SPN. Will choose the task with the shortest time left. Same issues as SPN.

5.2.6 Highest Response Ratio Next

$R = \frac{w+s}{s}$ w is wait time, s = service time = execution time. Choose the one with highest R. Again require knowledge of time, but provides a good balance.

5.2.7 Feedback

Timer interrupt. Decrease the priority of a process each time it is preempted. Does not require knowledge, but can cause starvation of long and old processes. Starvation can be fixed by giving older processes longer time slots.



5.2.8 Earliest Deadline First

The task with the lowest current deadline will be running. The priority of tasks changes with time.

5.3 Real-time scheduling and schemes

In RT deadlines are often critical, i.e. they must be met. So extra care must be used when scheduling.

5.3.1 Simple task model

Some assumptions are made

1. Fixed set of tasks
2. All tasks are periodic with known periods.
3. Tasks are independent
4. No cost of context switch.
5. Deadline = period.
6. Fixed worst case execution time.

5.3.2 Cyclic Executive

Hard code the scheduling. No tasks or processes, just function calls. Easy to share data and there are no race conditions.

Drawbacks of this method is that it is hard coding.

5.3.3 Fixed-Priority Scheduling (FPS)

Tasks are given static priorities and preemption with a timer is smart to use. Upon preemption the highest priority task that is ready will get to run. The priorities should be assigned such that they are optimal for the whole system.

Optimal priorities for periodic, independent tasks is **rate monotonic**: the shorter the period, the higher the priority.

5.3.4 Scheduling tests

There are test to see if a given set of tasks is possible to schedule. There are:

Necessary test Failed $\implies$ deadlines will be missed.

Sufficient Passed $\implies$ no deadline misses.

Exact Necessary $\wedge$ Sufficient

5.3.5 Utilization test

Can use if system is simple task model and rate monotonic. Is a sufficient test, but not a necessary.

$$U = \sum_{i=1}^{N} T \frac{C_i}{T_i} \leq N(2^{1/N} - 1) \tag{1}$$

U is utilization. N is number of tasks. T_i is period of task i. C_i is computational time of task i.

Alternative equation

$$\prod_{i=1}^{N} \left(\frac{C_i}{T_i} + 1 \right) \leq 2 \tag{2}$$

If there are tasks that are multiples of each other e.g $T_1 = 10, T_2 = 20$ then these together will only count towards N once.

5.3.6 Response-time Analysis

Calculates the worst case response time of a task when there are other tasks in the system interfering. This is an exact test. Has to be solved recursively. Stop calculating new values when there is no change. Check for interference from higher priority task.

$$R_i = C_i + B_i + I_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{3}$$

R_i is our current response time of task i. D_i is the deadline of task i. C_i is the compute time of task i. B_i is block time 6.1

5.4 Sporadic and Aperiodic tasks

Aperiodic tasks can be realeased at any time. Can no longer really use the Response-time Analysis.

Sporadic tasks have a minimum time which has to pass between releases. Can use this a the period as that is the worst case.

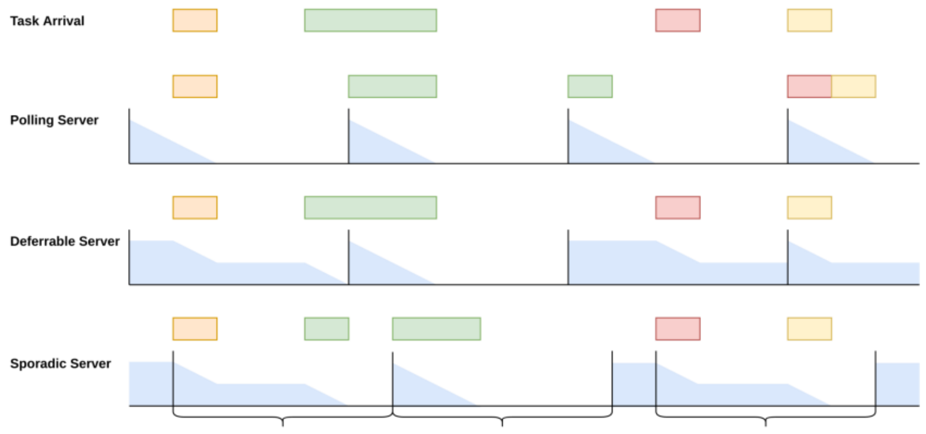
5.4.1 Execution-time Servers (ETS)

Method for handling aperiodic tasks. ETS runs periodically with a fixed budget. Lets normal tasks run whenever there is budget left.

A polling server just spends time regardless of tasks needing CPU time.

Deferable server only uses budget when there is a task running.

Sporadic server is same as sporadic, but the reset happens after a fixed time after the first use of a given budget.



5.5 Deadlocks and Starvation

A deadlock is when multiple processes wait for each other and thus all starve.

Three conditions has to be present for a deadlock:

Mutual exclusion Limited amount of processes can use resources at a time.

Hold and Wait A process can hold a resource while waiting for another.

No preemption No process can lose its resource access forcibly.

To prevent this you can either remove one of these conditions, which isn't always doable. Or you can program the order of resource requests to be the same for all processes wanting the same resources at the same time. If the system **detects** a deadlock it can be recovered in several ways.

1. Abort all deadlocked processes.

2. Return all locked processes to a checkpoint.
3. Abort one and one thread until the deadlock is resolved.
4. Preempt resources until the deadlock is resolved.

Deadlocks can be **avoided** in the following ways:

1. Process initiation denial
   (a) Prevent troublemakers from starting.
   (b) Check if max no. of resource users are already in the system.
2. Resource allocation denial - Only allow allocation if that leads to a safe state.

5.6 Worst case Execution time (WCET)

There are several levels of WCET. Can never really get the actual one. Can make statistical estimate.

Mixed cirticality is a measure of how bad it is if a task misses a deadline. Can be safety critical, mission critical, non critical or other.

5.7 Multiprocessor scheduling

Will only consider computers with multiple CPU cores, not clusters of computers or computers with multiple chips. The cores share main memory and usually OS. Not easy. Can run all RT tasks on one core and all others on another.

5.7.1 Core assignment

Static	Just put tasks in a CPU. No switches or overhead. Can lead to uneven load.
Global queue	A queue for all tasks. Whenever a core is idle it gets the next task in the queue. No idle cores when there is work to do. Will cause tasks to switch cores.
Dynamic load balancing	Assign tasks to cores and move them around according to load.

Can either schedule by master-slave configuration or peer configuration. On master slave the scheduler will run on just one processor. On peer it runs everywhere.

6 Priority inversion

When a high priority task has to wait for a lower one. This is OK when task A waits for task C to complete the usage of a resource. But let's say task B with a priority in between A and C blocks C from running. Then C cannot complete using the resource and thus A is blocked by B without B locking a resource A wants. A simple fix would be to give C the priority of A whenever C is blocking A's access to the resource and A is requesting it.

6.1 Priority Ceiling Protocols

Would it be better to give C the elevated priority without having to wait for A to request it? Maybe...

6.1.1 Original Ceiling Priority Protocol (OCPP)

Each task has a static priority which is assigned at creation. Each resource has a static value which is equal to the max priority of any task that may access it. Tasks also has a dynamic priority. The dynamic priority is max of its own static and any other static priority of a task it currently blocks. The trick is this: A task can only lock a resource if its dynamic priority is HIGHER than than the ceiling priority of any other locked resource. More efficient, but harder to implement.

6.1.2 Immediate Ceiling Priority Protocol (ICPP)

Each resource has a priority assigned to it which is equal to that of highest priority task that can access it. Whenever a task locks the resource it inherits the resource's priority. This will almost minimize the time a resource is locked. Easy to implement, but many priority changes.

6.1.3 Usage Function and Block Time

$$B_i = \sum_{k=1}^{K} usage(k,i)C(k) \tag{4}$$

usage is 1 for shared resources used other priority tasks. 0 otherwise.

The usage is the same for ICPP and OCPP.