NUMERO: 2338A ANNO: 2018

A P P U N T I

STUDENTE: Caldera Filippo

MATERIA: Operative system for ambedded system - Prof. Violante

OSES

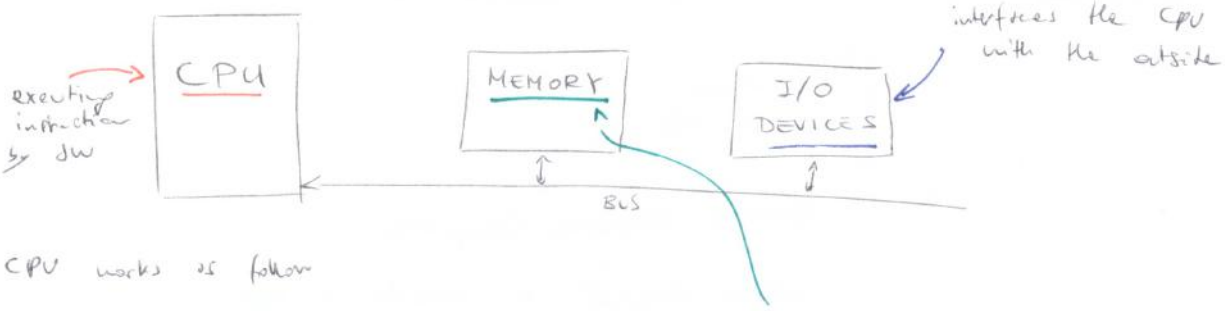
Operative Systems
for
Embedded Systems

– M. Violante –

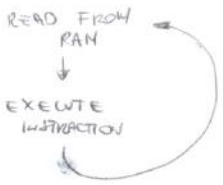
FILIPPO
CALDERA

© Proprietà riservata dell'autore - Digitalizzazione e distribuzione a cura del CENTRO APPUNTI - Corso Luigi Einaudi, 55 - Torino / Pagina 3 di 82

BASIC COMPUTER ARCHITECTURE



CPU works as follow



Computation Results are stored in registers

Store the instruction to be executed by CPU

There are two memory

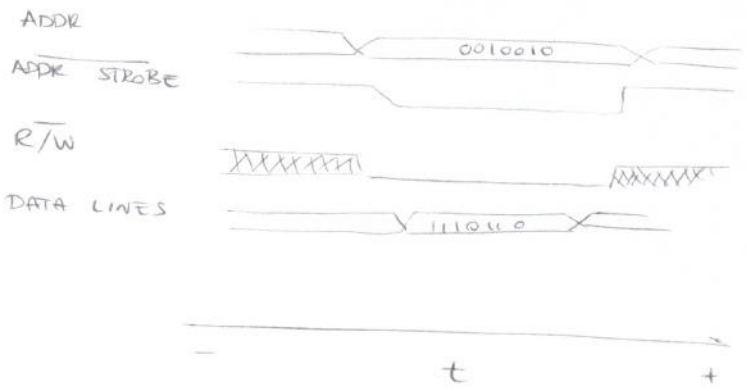
Volatile
=
works and stores data only if powered
↓
when computer is OFF the data is lost

Non-Volatile
=
store data also if power is OFF
↓
Data is kept anyway
↓
The first instruction when the machine is starting are stored here.

CPU starts any operation.

In case CPU needs info from RAM or I/O it performs the read/write cycle

Here the timing:



= giving address where to read } on Address Bus
= checks correctness } Control Bus
= W or R
= pass the data } Data Bus

Physically, the CPU is connected to the peripheral in 2 possible connection

PARALLEL BUS

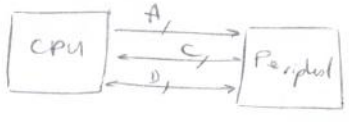
working principle:

t - Pass Address
check connection
t + Pass Data

⇓

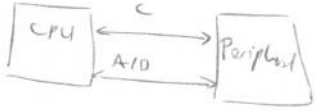
SLOW but data is transmitted in one slot.

NON - MULTIPLEXED



high number of b.l

MULTIPLEXED

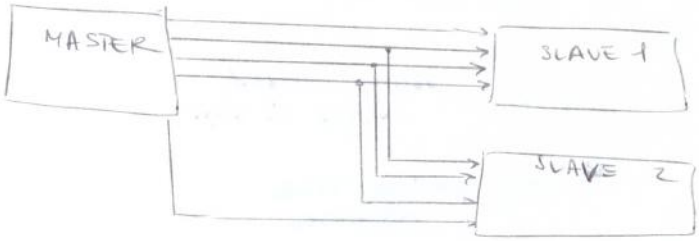


A, D shares the same wires

⇓

b.l reduced

SERIAL BUS = bus are shared with most peripheral



Communication is done over a single wire using Serial Protocols.

Shared bus are:

SCLK = Clock to Synchronize
MOSI = Master Out Slave In
MISO = Master In Slave Out

For each slave: SS = Slave Enable

only one slave is enable to talk with CPU using SS line.

Idea is let peripheral complete their own task and report the data only when CPU requires it.

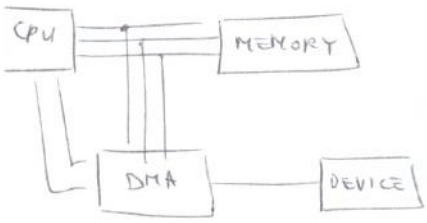
© Proprietà riservata dell'autore - Digitalizzazione e distribuzione a cura del CENTRO APPUNTI - Corso Luigi Einaudi, 55 - Torino / Pagina 7 di 82

When data needs to be transfer there are 2 ways

→ CHARACTER-BASED TRANSFER
= data transfered one word at time
⇓
Bandwith is limitted
⇓
Low efficiency

BLOCK-BASED TRANSFER
= data transfer as a cluster of bytes

It's a low-secure task the CPU transfers data
⇓
better have a dedicated HW doing transfer instead the CPU
=
Direct Memory Access (DMA)
=
responsible HW by CPU which send data (according to CPU instruction) from a address to another one
⇓
DMA acts the data transfer and allows the CPU to perform other task



CPU and DMA can have conflic over the bus
⇓
3 modes

Transparent
=
CPU has the total control over the bus (max priority)
DMA waits the bus to be not busy by CPU

→ Brust = DMA transfer data without interruption and CPU waits

Cycle Stealing = bus is used by CPU and DMA alterny the time usage

Operating System architectures

① FLAT

= huge function implementing applications, system interface. All linked together in order to have a single executable ⇒ No division between OS and Application

⇓

No protection to each other.

OS components are function which can be called by any application

If a malfunction occurs it can propagate without limits

⇓

Stucks the machine

Not reliable

② LAYERED / MONOLITIC KERNEL

System is divided in User space & kernel space

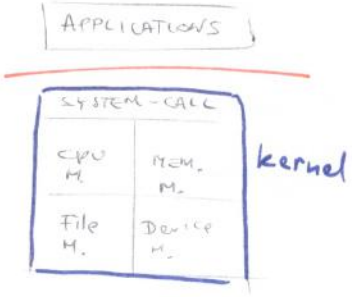
All services are delivered by kernel

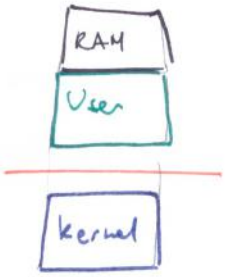
⇓

Application can't act on kernel space since address spaces are separated

⇓

More Robust Solution





kernel ad application talk though System-Call Interface

kernel is monolitic ⇒ All services are linked together

Versability is increased

If there is a new device drop the execution there is no need to reboot thanks to Loadable kernel Module

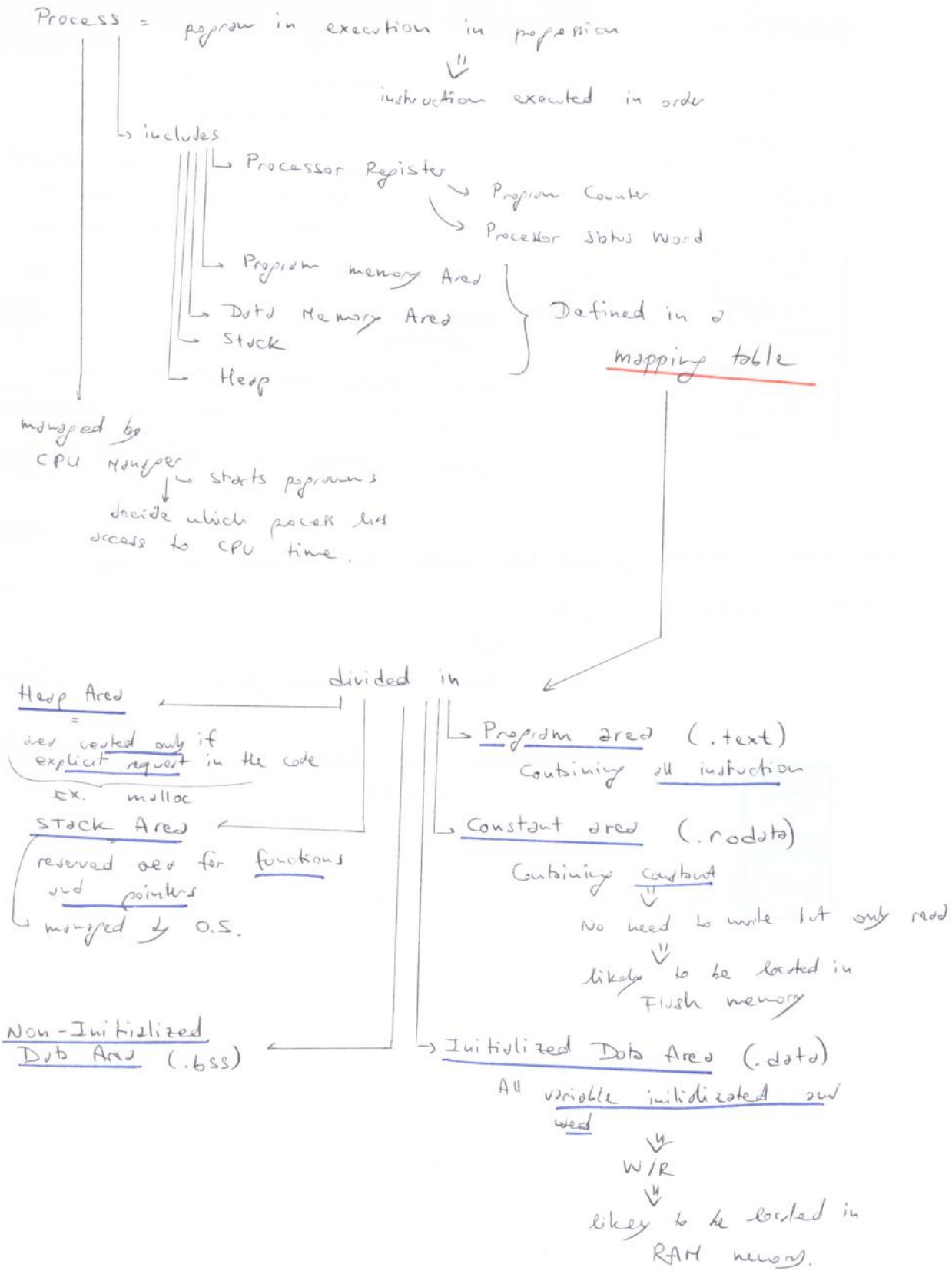
└ = part of code which can be "glue" with the executing one.

Since kernel is divided by Application ⇒ Fault Apps don't stuck the kernel

BUT Faulty kernel ⇒ Fault all the system

PROCESS MANAGMENT

L3 10/12/17



In an embedded system there are a lot of processes but only few CPU

⇓

In a CPU only one process is executed at time

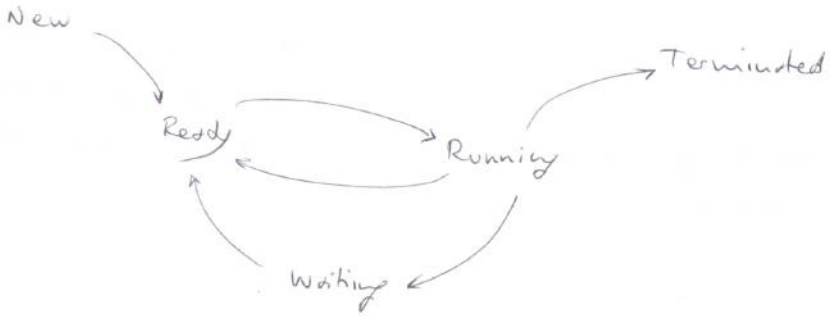
⇓

Need a PROCESS STATE = where the process is for the CPU time queue

can be

- New = process created, loaded in memory
- Running = process has CPU time ⇒ instruction executed
- Waiting = process lose CPU time because it's waiting some data for other devices
- Ready = process in the line for CPU time
- Terminated = execution finished

Process State Diagram (PSD)



To control the process along all these states a Process Control Block is created for each process

CRITICAL SECTIONS = piece of SW accedingo a shared exclusive resources.

Process use resource to move dtb / or processing

⇓

A resource can be use by any one process

⇓

In order to avoid conflict a resource can be

- SHARED = can be used by more tasks
- PRIVATE = only a particular process can use it
- SHARED EXCLUSIVE = shared but protected against concurrent access

Managed by CRITICAL SECTION

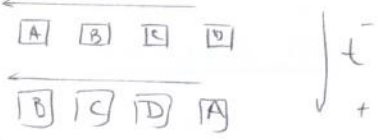
Priority is set.

why **SCHEDULING ALGORITHMS**

= criteria on which process execute along the ready queue.

few of them

- Round Robin
  - ↳ FIFO Based
  - All process are equal ⇓ No priority



Not very efficient

- Priority - based
  - ↳ use priority function p(T)

L4 10/13/17

MICRIUM µC/OS

- flat architecture
- ANSI C
- preemptive = using scheduler able to select which process to execute & capable of stopping running task

Good Safety-Critical System

CODE STRUCTURE

- Application Code = Application logic app.c, app.h
- CPU = specific function
- BSP = board specific code
- µC/OS-III
- µC/OS/CPU

} library functions.

In µC/OS a task = function whit infinite loop

```
void TASK( )
{
    Inizialization
    while(1) {
        TASK...
    }
}
```

∀ tasks ∃ · Task Cahol Block
· Task Stack

Task created using OSTaskCreate()

Each time a task is created there are

- OS_TCB = Task Cahol Block
- CPU_STK = stock Array

RACE PROBLEM SOLVING METHODS

ENABLE / DISABLE INTERRUPS

Disable interrupts in the critical section

only a process at time can use resources and free when it's done

interrupts are not predictable

No good for real-time

ATOMIC INSTRUCTION

Atomic instruction are not interruptible

BUSY WAITING

during critical section the shared resources is "lock down" and not accessible by other tasks

Need to use a boolean variable to lock

- Get Section
- Lock it
- Execute
- Unlock

EX.

```
do
while ( .... (*lock) == TRUE) ;   <- wait resource to be unlocked
CRITICAL SECTION                  <- do stuff when resource is free
lock = FALSE                      <- unlock resources when critical section is done
} while (TRUE)
```

Waiting

Waste of CPU Time

process is stuck doing nothing but still in execution

SEMAPHORES

idea is check the device availability and if the resource is unable then the running process is put in waiting state. when the resource is free, task is resumed.

based on 2 function

wait

DECREASE VALUE

IF VALUE < 0

SET PROCESS IN WAITING

RE-SCHEDULE

signal

INCREASE VALUE

IF VALUE ≤ 0

SET PROCESS READY

RE-SCHEDULE

In the task

while(1)
wait()
critical

LS 10/20/17

PROCESS SCHEDULING

based on → Constraints → Hard Real-Time
→ Firm Real-Time
→ Soft Real-Time

→ Task Type

All these parameters influence the scheduling

→ Periodic = infinite sequence regularly called

→ Aperiodic = infinite sequence called at not regular time

Possible to define minimum inter-arrival time

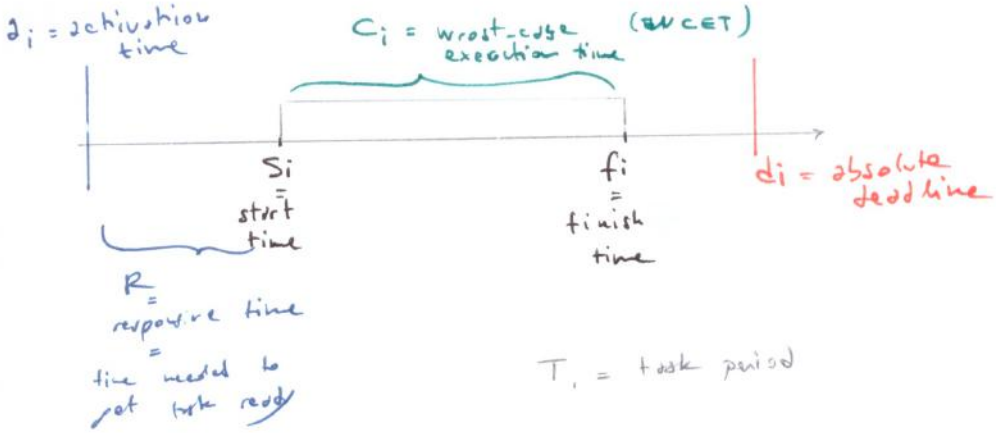
→ Sporadic
=
No possible to define min inter-arrival time

Few Assumptions ...

- # Tasks known.
- All tasks periodic
- Tasks Independent => No shared exclusive resources
- Fixed Worst-case execution Time known.
- System Processes negligible in time.

Basically, assumed all time properties are known.

... and Few Notations



PERIODIC TASKS SCHEDULING

↓

3 methods

- Time Scheduling
- Rate Monotonic
- Earliest Deadline First

• Time scheduling

↳ feasible scheduling without context switch

↓

Good for small system

=

A major cycle is repeated endless

divide time in

→ Major cycle = lowest common multiple of all task periods

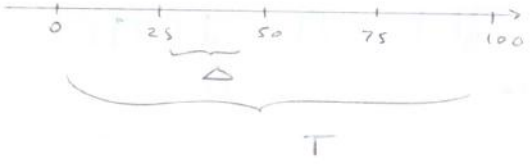
→ Minor cycle = greatest common divide of all task period.

If $\sum WCET|_{minor\ cycle} \leq$ Minor Cycle ⇒ Scheduling Feasible

EX.

$T_A = 25\ ms$
$T_B = 50\ ms$ ⇒ Major cycle = $T = 100\ ms$
$T_C = 100\ ms$

Minor cycle = $\Delta = 25\ ms$



CODE EX

- Task coded as function
- Defined Minor cycles counting task to be performed

```
Major(){
  while(1){
    minor_1();
    wait()
    mino_2()
    wait
    :
  }
}
```

EX

TASK	T	WCET
A	4	2
B	8	2
C	12	1

STEP 1: DEFINE T, ΔT

$\Delta T = GDC(4,8,12) = 4\,ms$ $\quad T = lcm(4,8,12) = 24\,ms$

STEP 2: GET TASKS FREQUENCY

$T_A = 4 \Rightarrow$ A executed 6 time skip T

$T_B = 8 \rightarrow$ B executed 3 tice skip T

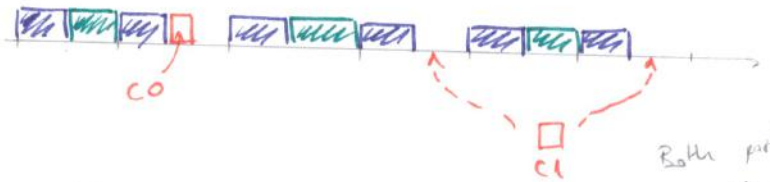
$T_C = 12 \Rightarrow$ C executed twice skip T

STEP 3: DRAW TIMELINE

STEP 3.1: TASK A

STEP 3.2: TASK B

STEP 3.3: TASK C



STEP 4: CODE

6 minor cycles $\Rightarrow$ 6 functions

```
M1(){        M2(){      M3(){      M4(){      M5(){      M6(){
  A();         A()        A()        A          A          A
  B();         C()        B()      }          B        }  C
}            }          }          }        }
```

EX.

TASK	T	WCET
A	4	2
B	8	1
C	12	1

STEP 1: CHECK IF FEASIBILITY SUSSISTS

$$\prod_{i=1}^{M} (U_i + 1) \leq 2 \quad \Rightarrow \quad \left(\frac{2}{4}+1\right)\left(\frac{1}{8}+1\right)\left(\frac{1}{12}+1\right) = 1.83 \leq 2 \Rightarrow \text{Scheduling Feasible}$$

STEP 2: GET PRIORITY

$T_A, T_B, T_C \rightarrow \Rightarrow T_C, T_B, T_A \Rightarrow$ A starts first

(Priority)

STEP 3: GET ΔT, T

$\Delta T = 4 \qquad T = 24$

STEP 4: GET FREQUENCY

$T_A = 4 \Rightarrow$ A has to be executed 6 times in T

$T_B = 8 \Rightarrow$ B " " " 3 times in T

$T_C = 12 \Rightarrow$ C " " " 2 times in T

STEP 5: TIMELINE

$U = \frac{2}{4} + \frac{1}{8} + \frac{1}{12} = 0.71$ ← Good!!

Let's assume to look for $X = WCET_A$ max but still having schedule feasible

$$\left(\frac{X}{4}+1\right)\left(\frac{1}{8}+1\right)\left(\frac{1}{12}+1\right) = 2 \quad \Rightarrow \quad X_{MAX} = 2.56$$

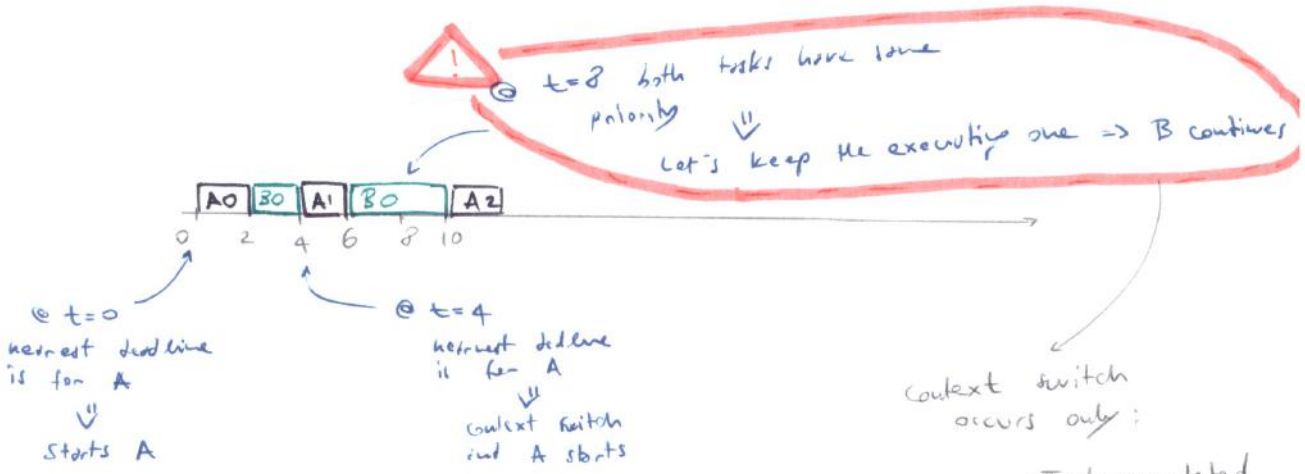
EX Given

	T	WCET
A	4	2
B	12	6

STEP 1: CHECK SCHEDULING FEASIBILITY

$$\sum \frac{c_i}{T_i} \leq 1 \Rightarrow \frac{2}{4} + \frac{6}{12} = 1 \leq 1 \Rightarrow \text{Scheduling Feasible}$$

STEP 2: TIMELINE



- Jackson

Earliest Due Date = tasks with lower deadlines are the first to be executed

Task need to finish first ⇓ Let's execute it first.

defined latency

L = amount of time ~~left~~ before deadline

$L_i = f_i - d_i$

f_i = finish time

d_i = deadline time

$\Rightarrow$ If $L_i < 0 \Rightarrow$ Good, no overtime

Good for Hard Real-time

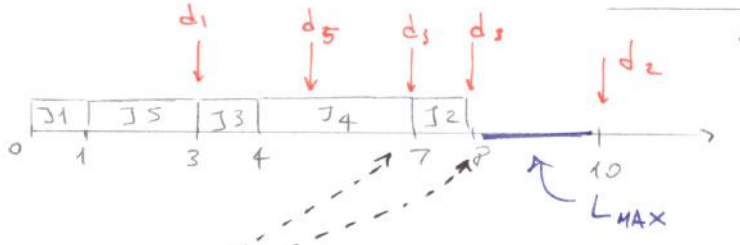
EX

	J_1	J_2	J_3	J_4	J_5
c	1	1	1	3	2
d	3	10	7	8	5

$\Rightarrow$ order of execution:

$J_1 - J_5 - J_3 - J_4 - J_2$

deadline



For example

$L_4 = f_4 - d_4 = -1$

$f_4 = 7$ ← depends by scheduling execution order

$d_4 = 8$ ← given

All tasks were activated at the very same time.

EX.

	a	d	WCET
A	0	5	2
B	0	3	1
C	0	10	5
D	0	6	3

STEP 1: CHOOSE ALGORITHM

$a = 0$ for all task = all tasks activated at the same time ⇒ tasks synchronous

⇓

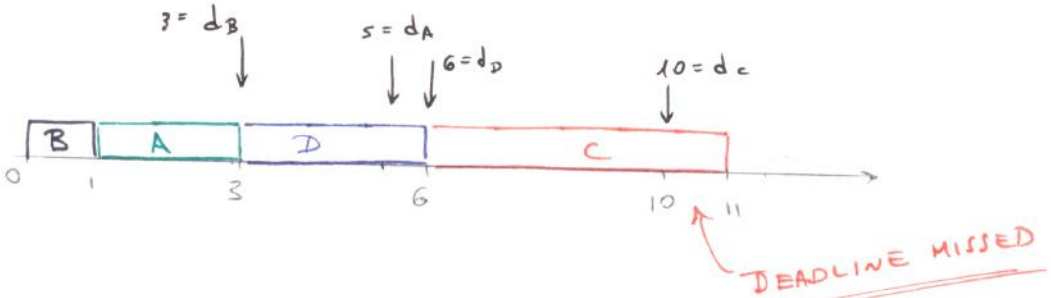
Jackson

STEP 2: DEFINE EXECUTION ORDER ACCORDING WITH DEADLINES

3 5 6 10 ⇒ B → A → D → C

STEP 3: TIMELINE

STEP 3.1: SHOW DEADLINES



$L_{MAX} = 11 - 10 = 1 > 0$

PERIODIC AND APERIODIC TASKS SCHEDULING

In real application there is a mixture of periodic and aperiodic tasks

- periodic → Time-Driven → Control to guarantee regular activities → Hard timing constraints
- aperiodic → Event Driven

→ Based on different algorithms

- Background Scheduling = execute aperiodic task only when there are no periodic tasks ready
- Fixed Priority Servers = Periodic tasks have fixed-priority and start at the same time.
  - Polling Server
  - Deferrable Server
- Dynamic Priority Server

Not covered by OSES

- Polling Server

L = special periodic task created to serve aperiodic tasks ASAP.

characterized by → T_s = period
→ C_s = capacity = amount of CPU time server can use uninterrupted

Concept idea:

Server scheduled as a periodic task using RM

If there are no aperiodic task ready $C_s = 0$ immediately ⇒ If no aperiodic request = Server lose all C_s

otherwise task consume server C_s.

∀ new period C_s is reloaded.

Feasibility

$$\begin{cases} \prod_{i=1}^{m} (U_i + 1) \leq \frac{2}{U_s + 1} \Rightarrow \text{Scheduling Feasible} \\ U_s = \frac{C_s}{T_s} \end{cases}$$

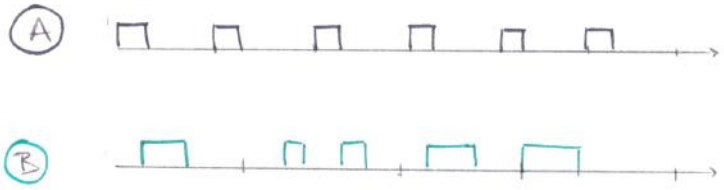
$$\begin{cases} P = \prod_{i=1}^{m} (U_i + 1) \\ U_s^{MAX} = \frac{2 - P}{P} = \frac{C_s}{T_s} \end{cases}$$

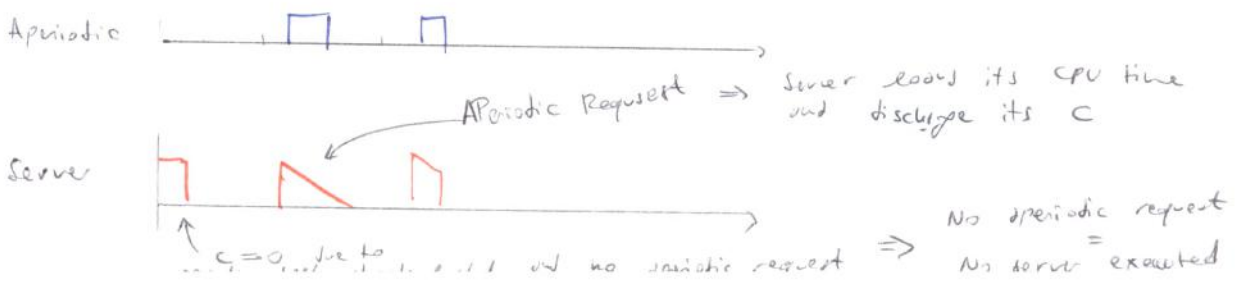
Feasibility contains only 2 parameters the other one is to choose according with the application.

Example

	C_i	T_i
A	1	4
B	2	6

Server
$C_s = 2$
$T_s = 5$



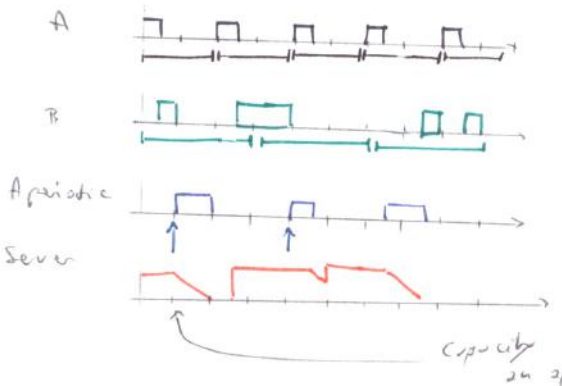


⇒ Server loses its CPU time and discharge its C

$C_s = 0$ due to ... and no aperiodic request ⇒ No aperiodic request = No server executed

- Deferrable Server

Idea is just like polling server but if there are no aperiodic requests C_s is NOT losted.



capacity is kept until an aperiodic task is ready

As soon an aperiodic task arrives it's executed according to RM scheduling.

Feasibility

$$\begin{cases} \prod_{i=1}^{n} (U_i + 1) \leq \dfrac{U_s + 2}{2U_s + 1} \\ U_s = \dfrac{C_s}{T_s} \end{cases} \Rightarrow U_s^{MAX} = \frac{2 - P}{2P - 1}$$

2. Minimize aperiodic task scheduling.

↓

Use deferrable server

↳ keeps C_s ⇒ Able to serve immediately aperiodic task.

STEP 1: DESIGN SERVER

STEP 1.1: FIND U_s^{MAX}

$$\begin{cases} U_s^{MAX} = \dfrac{2-P}{2P-1} \\ P = \left(1+\dfrac{4}{16}\right) = 1.25 \end{cases} \Rightarrow U_s^{MAX} = 0.5 = \frac{C_s}{T_s}$$

STEP 2.2: SET APERIODIC TASKS WITH HIGHER PRIORITY

$$\begin{cases} T_s \leq \min\{T_A\} \\ T_s \mid U_s = \max\{T_{APERIODICS}\} \end{cases}$$

T_s must be lower than the $\min\{T_A\}$ to have highest priority but it's also better to have T_s high enough to execute the largest aperiodic task

Good $C_s = WCET_B + WCET_C = 5ms$

$$0.5 = \frac{C_s}{T_s} \Rightarrow T_s = 10ms \leq 16ms$$

Choose $T_s = 10ms$ we have:

- $T_s < T_{min}$ ⇒ highest priority
- T_s guarantees execution of all periodic tasks

	T	WCET
A	16	4
B	/	4
C	/	1

B has min interval time $T = 8$ ms

C " " " " $T = 16$ ms

B, C aperiodic but for now it's known that between one call and another one there is a minimum time.

hard to deal with using levels

⇓

Aperiodic tasks are considered as periodic with T = min interval time

OVER ESTIMATION leading to waste CPU time.

⇓

	T	WCET
A	16	4
B	8	4
C	16	1

Solutions to priority inversion

- Non-Preemptive Protocol (NPP)
- Immediate Priority Ceiling (IPC)
- Priority Inheritance Protocol (PIP)

} Assuming RM scheduling

- **Non-Preemptive Protocol** = not allowing preemption during critical section

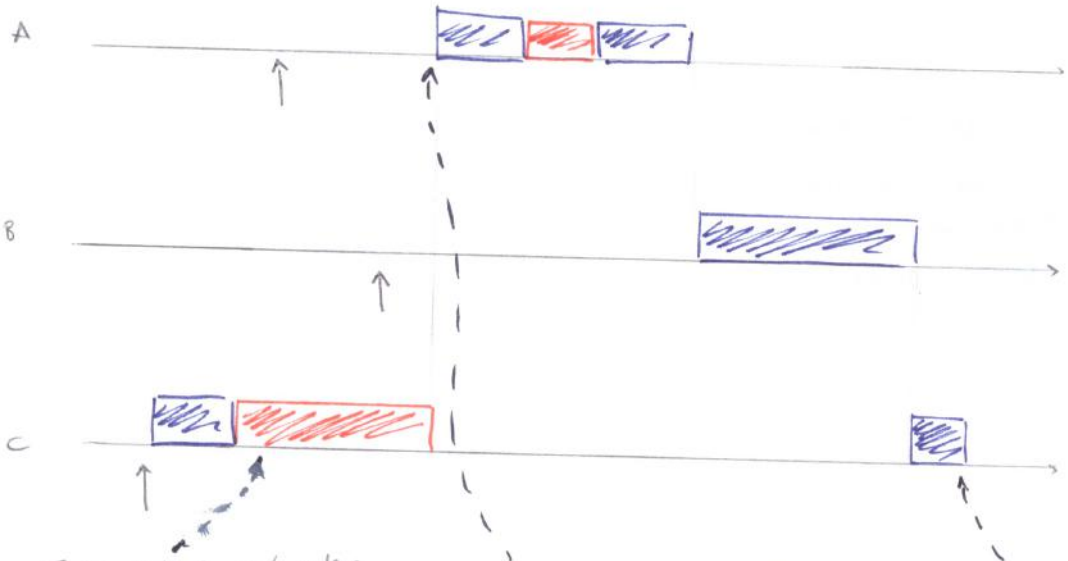
⇓

Task priority is promoted to the highest priority when executing critical section

EX

$P(A) > P(B) > P(C)$

▨ = critical section



C is called and enters critical section.

⇓

C has same priority of A

⇓

C keeps executing its critical section

when C terminates its critical section its priority is lowered

⇓

A has higher priority

⇓

A executed

C terminates its execution at the end due to lowest priority

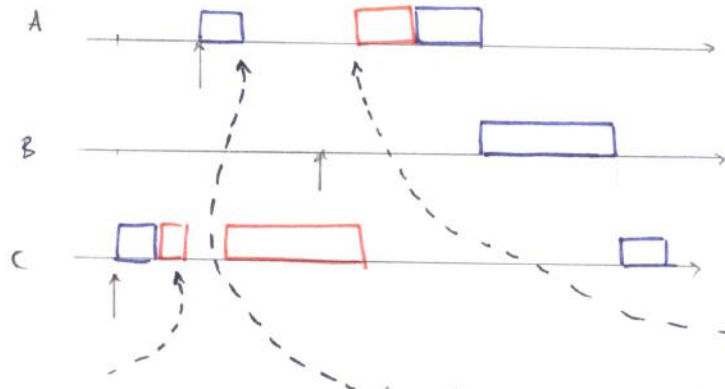
tasks are subject to priority inversion but at least we have a waiting time known and bounded!

⇓

Possible [illegible]

Priority Inheritance Protocol

When a task in critical section is blocking an higher priority task the its priority is promoted to the one which is blocky.



C is blocked since A has higher priority but isn't using resources yet

A can't use resources due to C is already using it.

⇓

C is blocking A

⇓

C promoted with same priority of A till critical section fully executed

When C has done with the resource A can access and execute its critical section.

Promotion only if task with higher priority is blocked.

MEMORY MANAGEMENT

L9 11/10/17

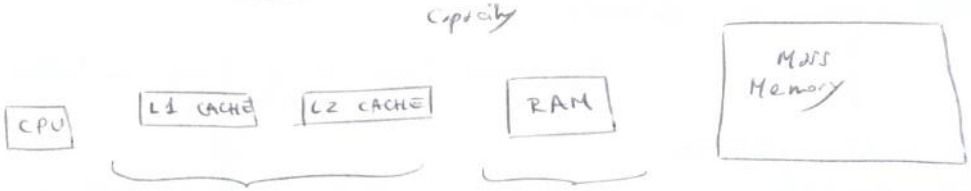
Memory = component storing informations

↓

in a system there are multiple type of memories

Speed ⟵ ⟶ Capacity

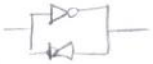
} When memory is close to CPU is smaller but faster



SRAM — DRAM

Today cycles are implemented inside CPU

storing memory using



storing memory using capacitors ⇒ Needs periodic refresh to maintain data

Volatile memory — Non Volatile

Main Memory (RAM) is splitted in partition

↓

managed by OS only

↓

OS has to provide suitable partition to program asking RAM space.

↓

accomplished by some techniques

- Static Partitioning
- Dynamic Partitioning
- Buddy System

} 1 for OS, more for user Process — size can be equal or different

Dynamic Partitioning

Allocations are ad-hoc ⇒ Memory partitions are sized to fit perfectly the process asking memory access

When a process has terminated its slot is freed

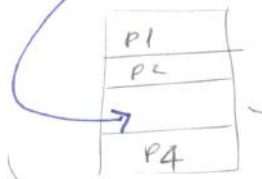
⇓

No internal fragmentation

Hole created

= free memory between used partitions

Need to manage holes



OS needs to known
- free partitions
- used partitions

an implementation is the

Buddy System

= allocating memory partition of size MUST be power of 2.

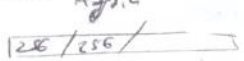
when a process is asking memory partition Buddy start a dicotomic division since the suitable partition is obtained.

ex. C asks partition of 120kB
Memory is 1MB large

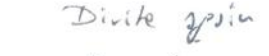
Buddy memory is 2

↑ 512kB too large

Divide again ⇐



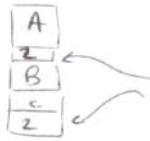
256 kB is too large ⟵ Divide again ⟶



if $\frac{128}{2} = 64 < 120$ ⇒ Buddy assigns to C a 128kB size partition.

In the Buddy System method can occurs

- Internal Fragmentation
  - Difficult to have a process requiring a 2^N sized partition ⇒ Free, not used but reserved
- External Fragmentation
  - Process can't be loaded because there is no partition even though the sum of the holes can fit it



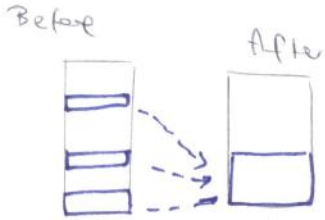
Total Hole memory = 2+2 = 4

but no possible to load a process

to solve this problem it's used a memory compaction

=

basically all the holes are moved to the end of the memory forming a bigger free memory slot



Not possible to load anything due to low memory sized

Possible to allocate process

EXERCISE

Given the tasks

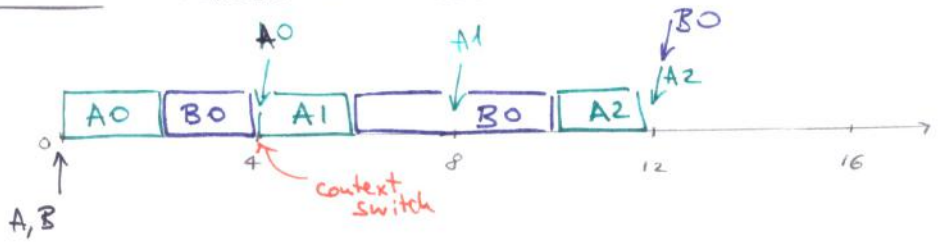
	T	WCET
A	4	2
B	12	6

task scheduled in EDF

· a task 1024 bytes are allocated

Find memory allocate which minimize fragmentation.

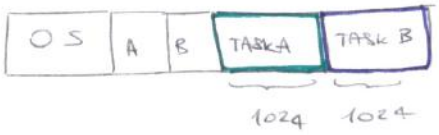
STEP 1: SCHEDULING EDF



STEP 2: MEMORY

Now there is preemption due to context switch @ t = 4

need 2 partition



STACK SHARING

For each task a stack memory is created

If • # stack ↑

• Embedded system with low memory available

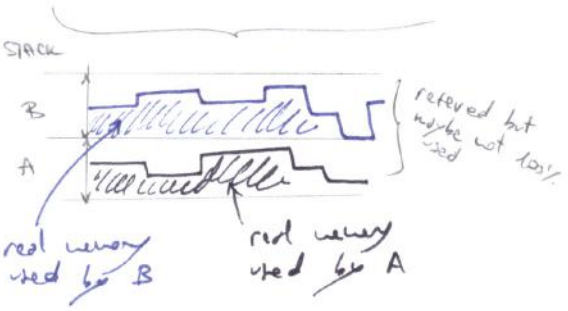
⇒ Need large amount of memory which is not provided

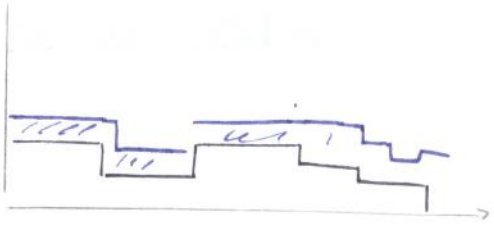
⇓

stack sharing used

=

Stack is single and shared with tasks only if tasks with different priority are present





Process with higher priority have task starting up the lower tasks

⇓

There is stack access rules to prevent contamination } priority + LIFO

ex.

Let's have 100 Tasks

1 Task requires 10kB of stack ⇒ • Using one stack for each task = 1 MB of stacks

10 priority level

Saved a lot of memory } • Using shared stack = 10 kB · 10 = 100 kB

L10 11/29/17

MASS MEMORY MANAGMENT

used for keeping data for unlimited time

need to be stored in a logical manner and be human readible

Need File Concept ⇐

File is a logical contiguous address space } = File = a container

All this provided to end user by Mass Memory Mngment

Types of Files
- Data
- Program → need OS to be read

File Structure
- None ⇒ stream of data
- Simple Record Structure
- Complex Structures

} Data organization for the user

Decided by
- O.S.

end user through O.S.

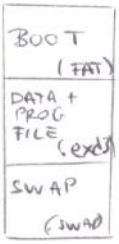
File Attribute
- Name
- Identifier = int number to have unique identification for O.S.
- Type
- Location = where file is located
- Size
- Protection

} For Human

} For O.S.

Two Partitions can belong to the same disk but are logically different

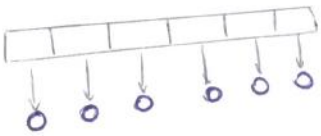
EX. LINUX



Some Volume with partitions → with different file system

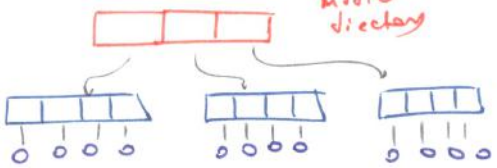
Directory Organization

Single Level



Files

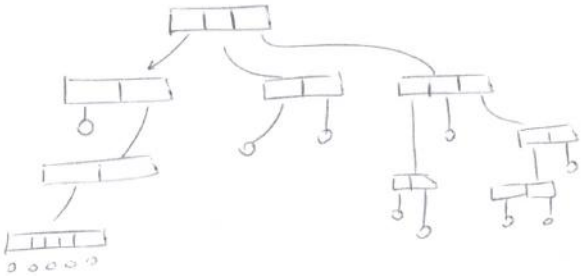
Two-Level



Need path to get a file

No possible to have sub-directory in directories

Tree-Structured



Possible to have directory in directory

If complexity ↑ ⇒ Need to store info about structure.

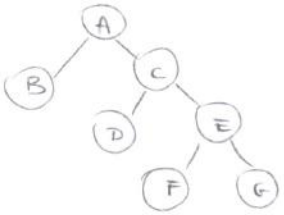
Memory Technology = how memory are physically implemented

- FLASH
  - non-volatile => hold data even when unplug to power
  - organized using sectors = smaller quantity possible to write
  - types
    - NOR
      - Line
      - low power
      - linearly accessible memory
      - Individually clearable bits
      - Reset clear only blocks, not entire memory
    - NAND
      - Bit Line
      - x8
      - 8 transistor making 8 bit lines
      - divided in 512 pages
      - not linearly accessible
      - Uniform Interface 8 bit

EXAMPLE

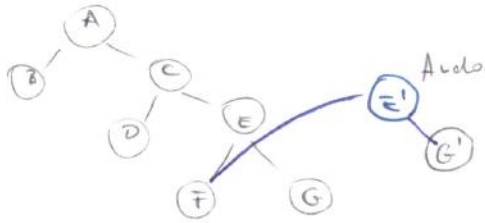
STEP 1: G' NEW

Anchor = (1, A)



contains info to be modified

STEP 2: CREATE NEW G' PARENT, E'

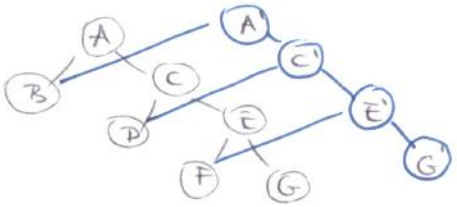


Anchor = (1, A)

need since user would access the newest content G'

Creating E' because is possible to notify E

STEP 3: CREATE NEW PARENT 'TILL THE ROOT



(1, A) = Anchor

Step 2 repeated 'till the root.

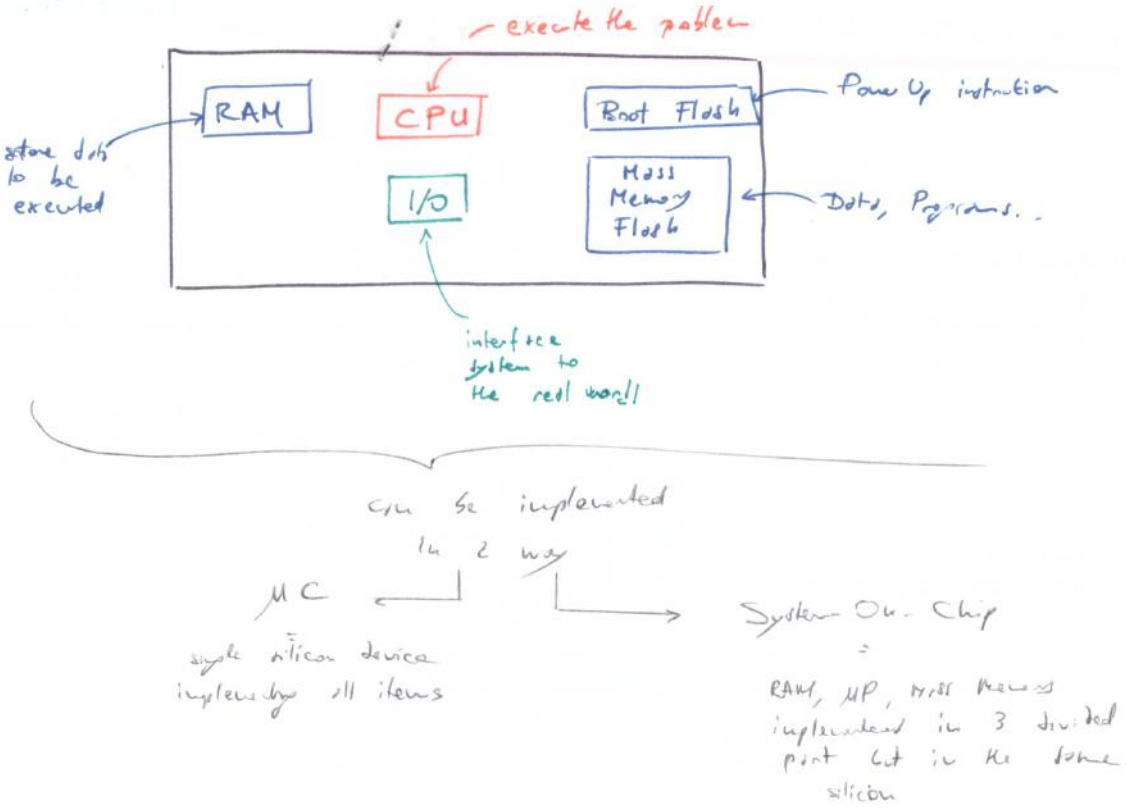
STEP 4: UPDATE ANCHOR NODE

Anchor = (1, A) / (2, A') ← New root

Not deleting anything ⇒ Using LogFS requires a lot of memory

⇓

Good for low frequency memory update



CPU can be implemented in 2 way

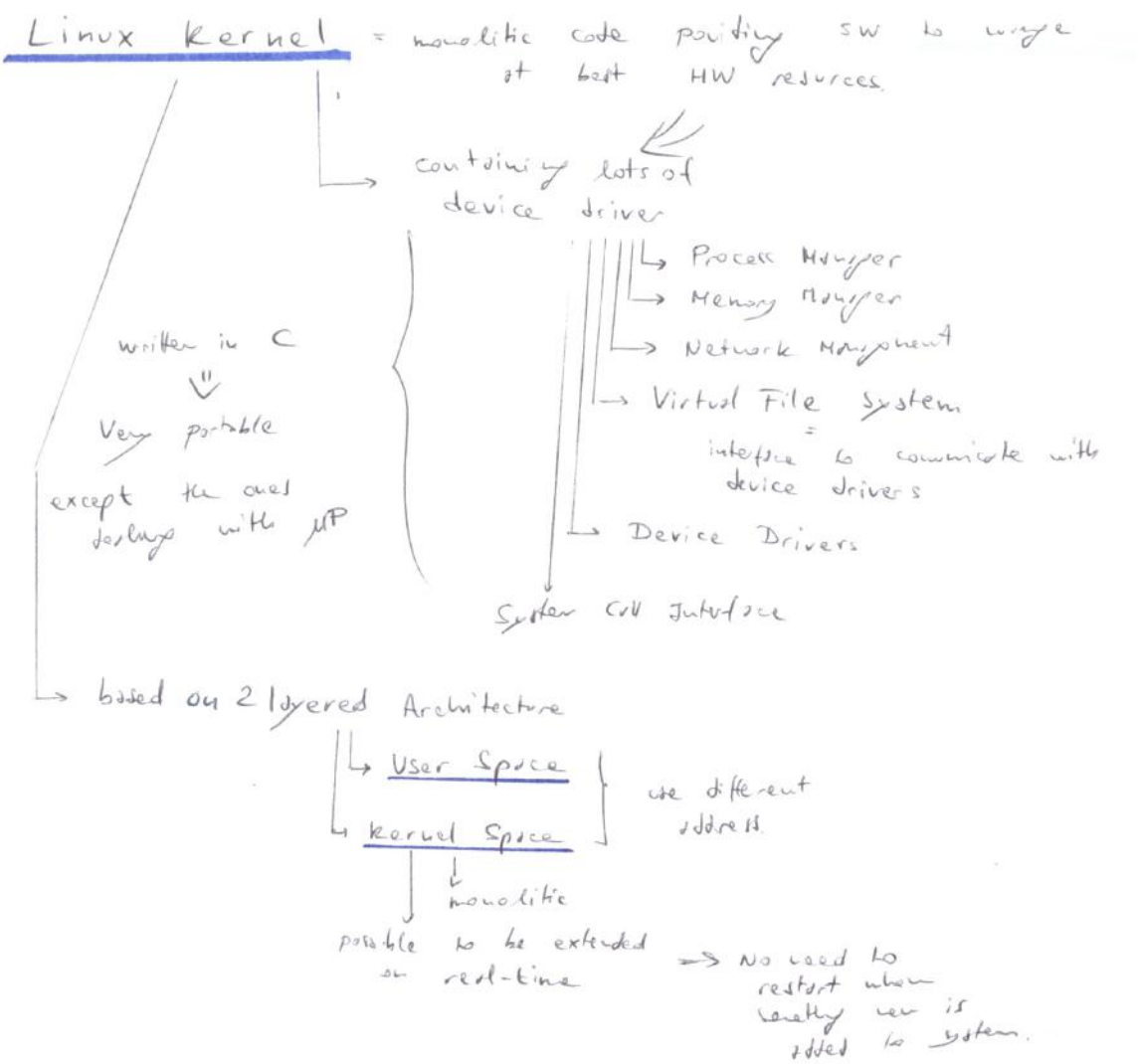
μC

single silicon device implementing all items

System On Chip

=

RAM, μP, Mass Memory implemented in 3 divided part cut in the same silicon


Linux Kernel = monolitic code posting SW to wrap at best HW resources
containing lots of device driver
Process Manager
Memory Manager
Network Management
Virtual File System
= interface to commicate with device drivers
Device Drivers
System Call Interface
written in C
Very portable
except the part dealing with μP
based on 2 layered Architecture
User Space
Kernel Space
use different address
monolitic
possible to be extended on real-time
No need to restart when newly ver is added to system.

Operating System for Embedded Systems

Theory exam simulation

Duration 60 minutes

1. **(5 pt)** Given the definition of process:
   a. List the different sections that define a process in memory; **stack, heap, bss, data, text**
   b. Given the following code fragment, identify the memory sections the different memory elements belong to:

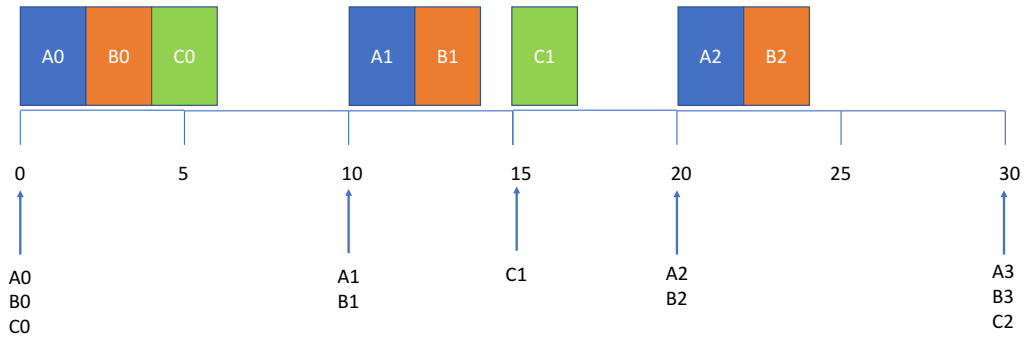
```
#include <studio.h>

int alfa = 25; → data
int beta; → bss

int dummy_function( int x → stack, int w → stack) → text
{
   int gamma; → stack;
   static int delta → bss;
   ....
}
```

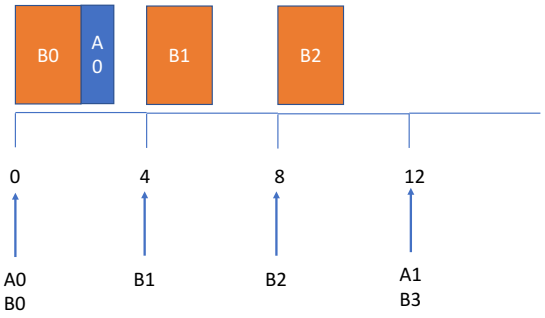
2. **(5 pt)** Assuming a task `T` is running, and the counter of a semaphore `S` is `0`, illustrate the process state diagram change in case `T` performs the operation `pend(S);`
   Running→Blocked if S==0, the task remains blocked until another task performs post(S).

3. **(5 pt)** Assuming the following periodic, independent, tasks must be scheduled on a single processor system:
   a. Propose a feasible scheduling when x=2→**using RM we have:**

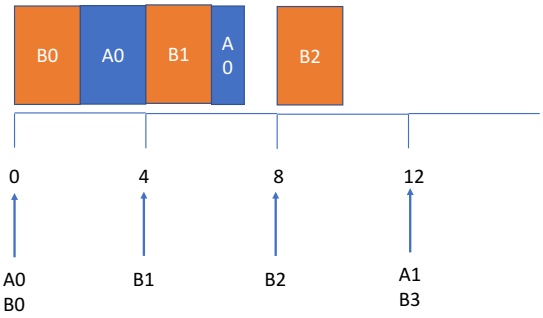


Task	Period [ms]	WCET [ms]
A	10	2
B	10	2
C	15	x

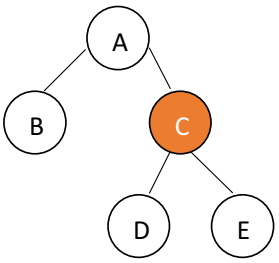
   b. Find the greatest value of x for which an RM scheduling is feasible; **given the above scheduling the maximum value of x such that RM is feasible is x=7**

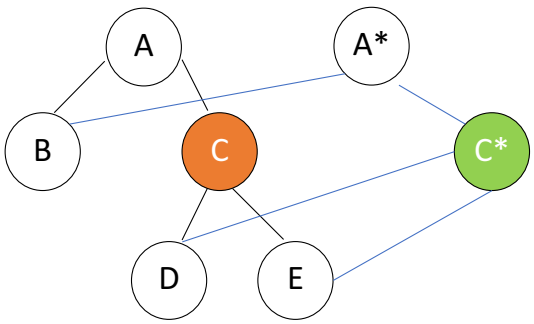


b. Assuming x= 3 and RM scheduling is used, compute the number of memory partitions needed for satisfying the memory needs for the tasks; → **due to the preemption needed to schedule A0 and B0, we need 1 partition for the code of A and 1 partition for the data of A, 1 partition for the code of B and 1 partition for the data of B, in total we needed 4 partitions.**



7. **(5 pt)** Given a logfs file system with the following structure, and anchor node (A, 1) draw the file system resulting from a modification to the node C.





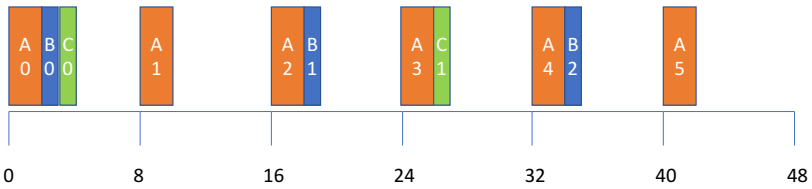
with anchor node (A*,2)

14. **(5 pt)** Assuming the following periodic, independent, tasks must be scheduled on a single processor system:

Task	Period [ms]	WCET [ms]
A	8	2
B	16	1
C	24	x

a. Assuming x=1, compute a feasible timeline scheduling if any.

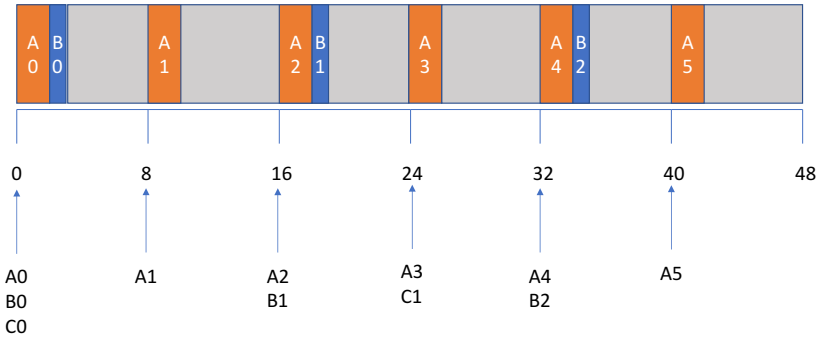
Given the period of the tasks, GCD=8 and lcm=48, thus the minor cycle will last for 8 ms, and 6 minor cycles will compose the major cycle. The resulting scheduling will be produced:



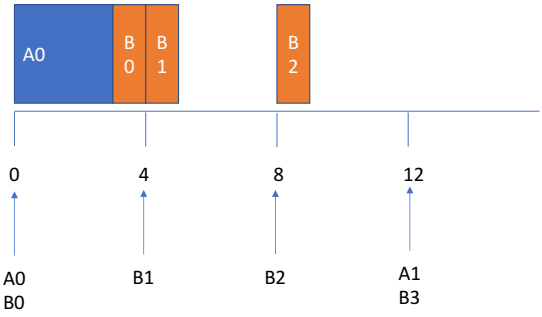
The scheduling is feasible as in each minor cycle the sum of the WCET of the tasks is always smaller than the minor cycle duration.

b. Find the greatest value of x for which an RM scheduling is feasible;

Given the period of the tasks, the maximum value of x that makes RM feasible can be computed as min(x1, x2), where x1 is the CPU available from 0-24 ms, and x2 is the CPU time available from 24-48 ms



$x1=24-3*WCET_A-2WCET_B=24-6-2=16$ ms
$x2=24-3*WCET_A-WCET_B=24-6-1=17$ ms
Therefore x = 16 ms, resulting in the following scheduling



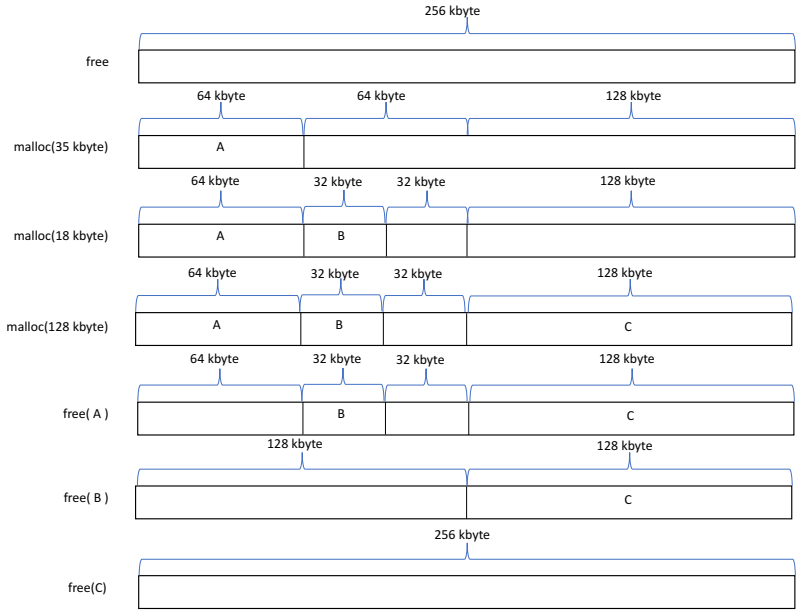
16. **(3 pt)** Describe the concept of stack sharing.

Stack sharing consists in allowing tasks of the same priority level to share the same memory portion for implementing the task stack. As a result, each priority level has a single stack space, rather than having a dedicated stack space for each task in the system.

17. **(5 pt)** Assuming a system sets available 256 kbytes of RAM, given the following sequence of memory operations, draw the different memory status when a Buddy allocator is used upon completion of the execution of each operation:

    A=malloc(35 kbytes); B=malloc(18 kbytes); C=malloc(128 kbytes); free(A); free(B); free(C);

The memory evolution is the following:



PERIODIC TASKS SCHEDULING

$$u_i = \frac{C_i}{T_i}$$

$$u = \sum_{i=1}^{N} u_i = \begin{cases} > 1 & \text{Not Consistent} \\ \approx 1 & \text{Border Line} \\ \ll 1 & \text{Good} \end{cases}$$

→ TIME SCHEDULING

$\Delta = CDC(T_i)$

$T = lcm(T_i)$

if $\sum_{i=1}^{N} WCET_i \leq \Delta \Rightarrow$ Feasible

→ RATE MONOTONIC

$T_i \uparrow \Rightarrow P_i \downarrow$

if $\Pi(u_i + 1) \leq 2 \Rightarrow$ Feasible for Sure

→ EARLIEST DEADLINE FIRST

if $\sum \frac{C_i}{T_i} = \sum u_i \leq 1 \Rightarrow$ Feasible

APERIODIC TASKS SCHEDULING

→ JACKSON (EDD)

= execute the one with the closest deadline

{ if all task have same arrival time

→ HORN (EDF)

= Like Jackson but with different arrival time

APERIODIC + PERIODIC TASKS SCHEDULING

→ BACKGROUND SCHEDULING

= Aperiodic tasks executed when CPU has spare time. Periodic task scheduled with RM.

→ SERVERS

↓ Parameters

$u_s = \frac{C_s}{T_s} = \frac{\text{capacity}}{\text{Period}}$

→ POLLING

If $P = \prod_{i=1}^{n} (u_i + 1) \leq \frac{2}{u_s + 1} \Rightarrow$ Feasible

$u_s^{MAX} = \frac{2 - P}{P}$

→ DEFERRABLE

If $P = \prod_{i=1}^{n} (u_i + 1) \leq \frac{u_s + 2}{2u_s + 1} \Rightarrow$ Feasible

$u_s^{MAX} = \frac{2 - P}{2P - 1}$