MICROPROCESSOR SYSTEMS (IAS0430)

Department of Computer Systems
Tallinn University of Technology

THE PROCESS

- The primary function of an operating system **is to provide an environment where user programs can run.**
- The operating system must provide a framework for:
  - **Program execution**
  - **A set of services: (file management etc.)**
  - **An interface to these services (API)**
- On a **multiprogramming system**, the operating system must also provide mechanisms to make sure that the programs loaded into memory and/or are executing at the same time, **do not interfere with each other**.
- This **restricted form of program execution**, which has access to the services of the operating system, is known as a ***process***.
  - **a process is a sequence of computer instructions executing within a restricted environment.**
    - **What makes a process?**
      - Think of what we learned in the CPU, kernel and user modes, and OS classes!

THE PROCESS

- A **process** is a sequence of computer instructions executing within a restricted environment.
  - We also know it as a Job! Or a program!
  - But what if we want to break a job into multiple processes to finish its execution faster!
    - If we have **4 CPUs** and only **one job**, if we divide the **job into 4 processes** and run those **4 processes at the same time using the 4 CPUs**, we will surely finish its execution faster… right? ---- obviously
  - **In reality, all modern computers divide Jobs into multiple processes.**
    - More so, processes are further divided into ***threads*** *– not for this course*
  - In order for a Process to execute, it needs an **environment** for execution!
    - Such environment is called **Process Context**
  - The process context consists of **three parts:**
    - The Address Space
    - Reserved Registers
    - System Calls

THE PROCESS

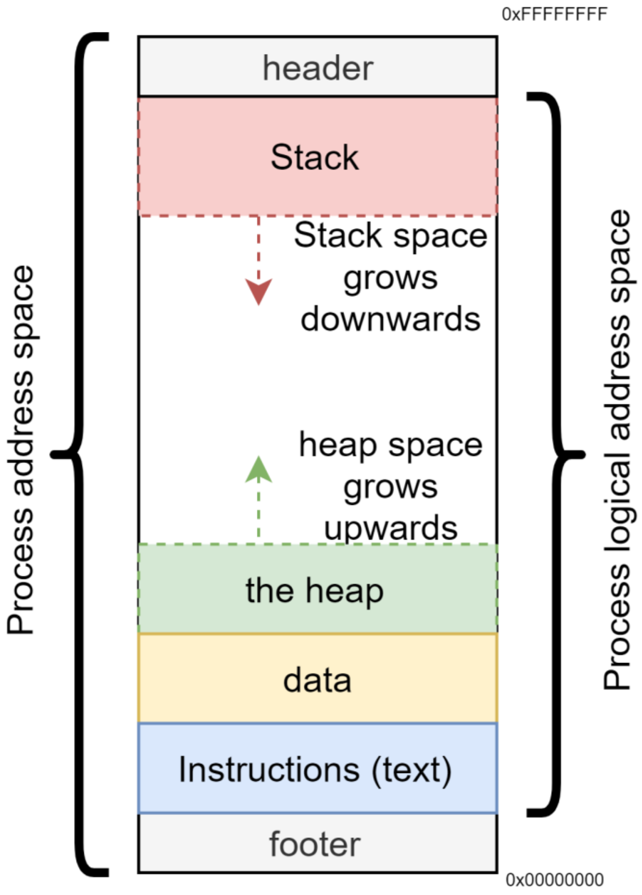
- **The Address Space:**
  - Is a reserved set of memory locations where we store:
    1. The **machine-code instructions** required by process
    2. A **data area** for the process' **global variables**
    3. A **stack area** for the stack frames which contain the **process' local variables**.

  - The address space is **protected,** so that **other processes cannot interfere** with the execution progress of the process. Preventing other processes from:
    1. Changing any or all instructions needed to execute the process
    2. Accessing local variables of the process, as those are private information
    3. Changing/updating the global variables of the process

  - Only the process and the OS can access its address space.
    - Other processes can access an address space if, and only if, they run in kernel mode, allowed by the OS, and have certain privileges to do so.

THE PROCESS

- **The Address Space:**
  - The address space is divided into areas:
    - The **instruction space** where process instructions are stored
      - Includes a binary image of the process.
        - The **binary image** is the process’ executable code – instructions
    - The **data space** where **static variables** are stored
    - The **heap space** where the **dynamic memory allocation** occurs.
      - The heap can grow as long as the process needs and as long as the process address space is not exceeded.
    - The **stack space** where private local variables are stored
      - The stack can grow as long as the process needs and as long as the process address space is not exceeded.
    - The **header**: OS reserved kernel space. Only OS can access.
    - The **footer**: Includes information on the size of the address space, ID of the process and other information

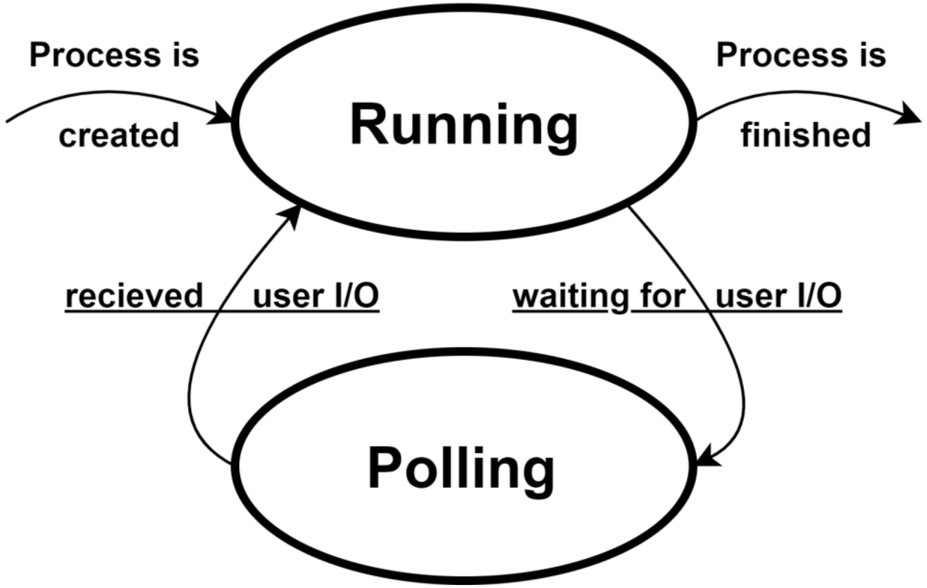


THE PROCESS

- **Reserved Registers:**
  - Is a set of **CPU registers are exclusively used by this process** in order to finish operations (addition, subtraction, comparison, etc.) quickly instead of needing access to slower memory (cache or RAM).
  - These registers are also used for invoking syscalls.
    - By loading a register with a specific value, the CPU know what operation to perform for that process in kernel mode.
- **System Calls**
  - As we know, a system call is a special instruction called the trap instruction.
  - A process needs access to syscalls in order to access the different OS services.
    - Once a process needs a service, it uses one of the reserved registers to store a specific value, and uses the trap instruction to tell the CPU which register that value is stored in.
- Once these three parts are available, **a process context exists.**

THE PROCESS

- **Process states and process models:**
  - Since a process is an executable set of instructions, there need a way to determine if a process is **ready for execution**, if a process is **waiting for data to arrive from I/O**, if a process is **being executed**, and if a process is **finished**.
  - For this, there are many models that indicate which state a process is in:
  - **The embedded system model**
    - Embedded system only have one process running at a time.
      - This means that a process is either being executed or waiting for I/O
    - As a result, a state diagram of an embedded system process stated would look something like this:
    - The **polling state** is when the process needs input from the user to continue.

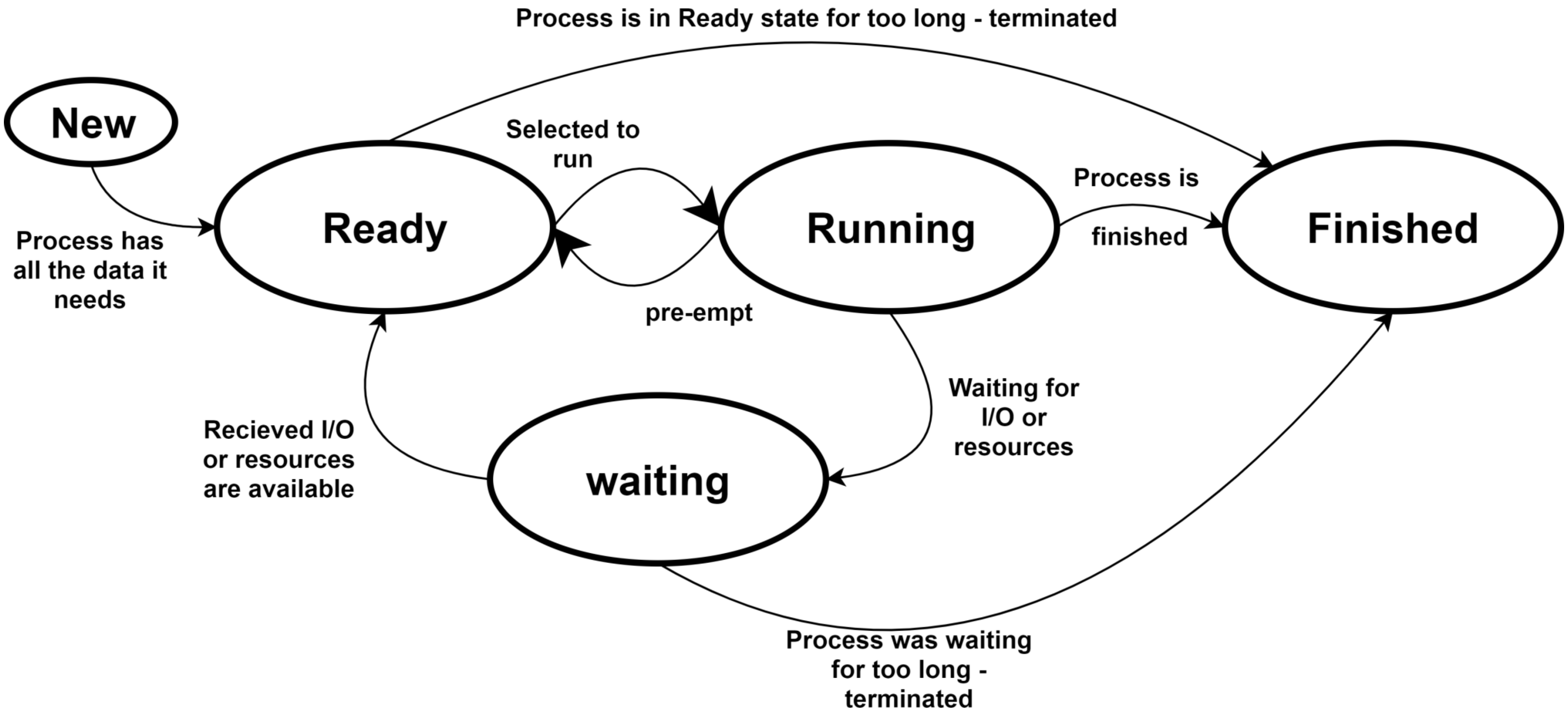


THE PROCESS

- **The batch system model**
  - Most systems can process multiple processes at the same time.
  - This can be done by creating a **batch of processes and managing them** one at a time.
  - Processes that **have all the data they need to run**, can be queued in a **ready** list in order to run them as soon as the CPU can.
  - While running, **if a process required I/O from the user or needs additional recourses** to free up, it can be put into a **state where it waits** until it gets the data it requires to resume running
  - Once a process has received the data it needs, it can be put into the **ready** queue again to be ran once the CPU can.
  - If a running process is taking too long to run, the process manager will create a what is called a **pre-empt**, the process is put back to the ready state, allowing fair use of the CPU time among all the processes.
  - Once a process is done, it is moved into a **finished** state, allowing the MMU to deallocate its Address Space.
  - If a process is either waiting for I/O or resources or is ready to execute for **too long**, it is also **terminated** and put into a finish state.

THE PROCESS

- **The batch system model**



THE PROCESS

- **Process Control Blocks:**
  - Now that we know what states a process can have, we need to keep track of that somehow!
  - This is done using the **Process Control Block (PCB).**
    - The **Process Control Block** is a collection of information regarding a process. The OS uses it to keep track of several parts of the process:
      - **Contains the process' memory space** – prevent deallocation of that space and protects it from other processes
      - **Save copies of the CPU registers** in the PCB.
      - Save information about any other resources used.
      - Mark the process' new state.
      - If in Waiting state, **record which I/O operation the process is waiting for**, so when I/O completes, the process can be moved back to Ready state.
    - The **PCB** has many elements stored in it, those elements can be put in the following categories:

THE PROCESS

- **Process Control Blocks:**
  - **Machine independent:**
    - Process ID.
    - Process state.
    - The process' priority.
    - … etc.
  - **Machine dependent:**
    - Information on Address space.
    - Copies of registers
    - …etc.
  - **Statistics:**
    - Total time of execution.
    - Estimated time of finish.
    - Total of time the process was in each state.
    - …etc.

THE PROCESS

- **Process scheduling:**
  - Now that we know what are the possible states of the batch system, there must be a way for the system to decide which process should be selected to run or which process to be pre-empted!
  - This is the function of the process (or processor) scheduler
    - The process scheduler is an OS service that manages the CPU time by deciding what processes are selected to run and when they are pre-empted.
- **Scheduling decisions**
  - Long-term scheduler decides which jobs/processes are to be admitted to the ready queue (in main memory); and dictates what processes are to run on a system, and the degree of concurrency to be supported at any one time.
  - Medium-term scheduler temporarily removes processes from main memory and places them in secondary memory (such as a hard disk drive) or vice versa (swapping).
  - Short-term scheduler (CPU scheduler) decides which of the ready, in-memory processes is to be executed (allocated a CPU) after a clock interrupt, an I/O interrupt, an operating system call or another form of signal.

THE PROCESS

- **In general, there are two types of scheduling schemes**
  - Non-pre-emptive scheduling – a job/process is running from the beginning to the end without interrupts (unless interrupts are in the code).
    - First Come, First Served (aka FIFO) scheduling
    - Shortest Job Next (SJN) scheduling
    - Priority scheduling
  - Pre-emptive scheduling – a running job/process may be interrupted and put into wait state depending on different events/priorities
    - Shortest Remaining Time (SRT) scheduling
    - Fixed priority pre-emptive scheduling
    - Round-robin scheduling
    - Multilevel queue scheduling
    - Work-conserving scheduling

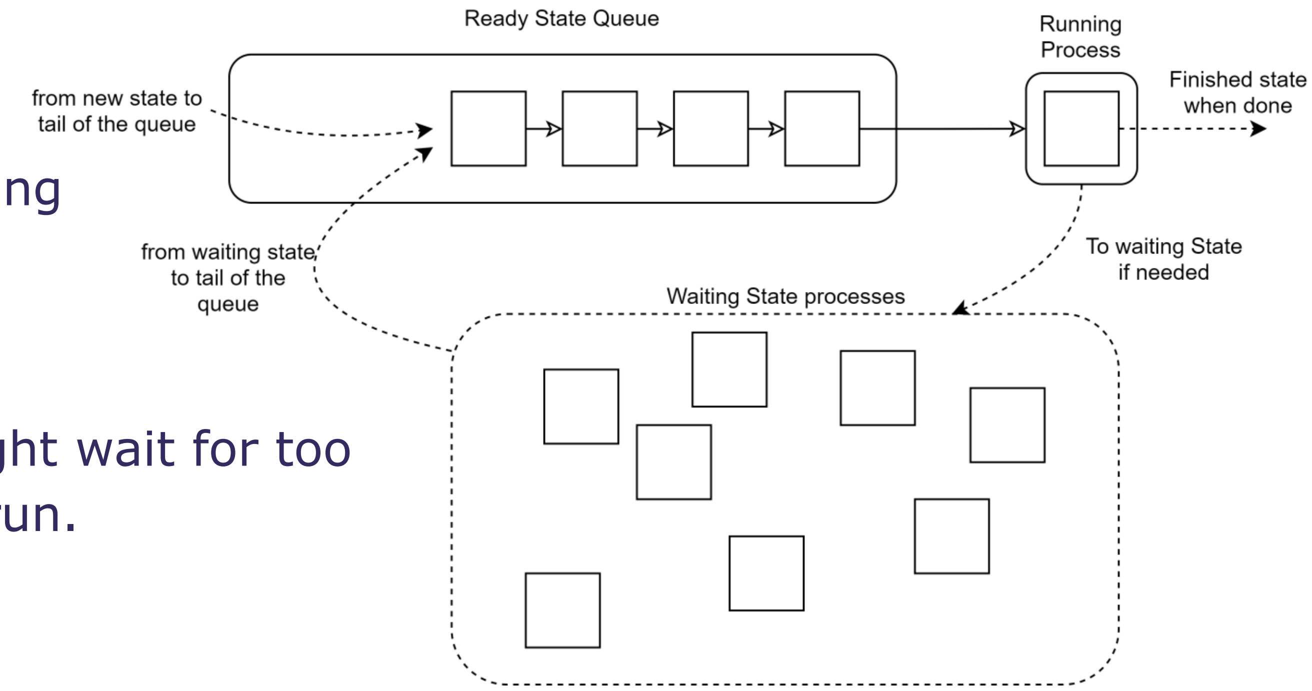
- See also https://en.wikipedia.org/wiki/Scheduling_(computing)

THE PROCESS

- **Process scheduling:**
  - **Non-pre-emptive** schemes: Where a process is never moved back to the ready from the running state – **a process in running state is never stopped unless an I/O or a resource is needed: it will keep running until it is finished.**
    - This can causes **resource hogging**.
      - This problem occurs when a job takes a very long time to finish and forces the MMU to allocate a large number of resources for it.
      - This prevents other processes from starting or arriving to a ready state.
    - **These schemes can cause significant performance degradation if not used properly**

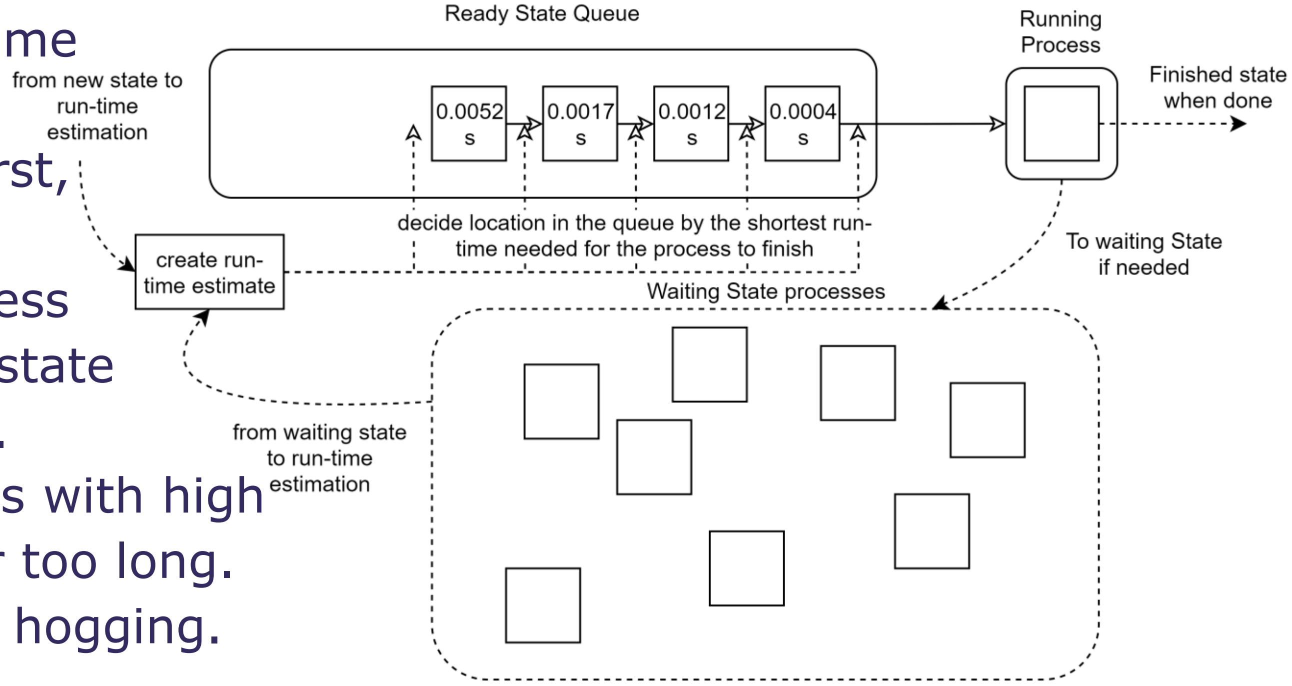
THE PROCESS

- **Process scheduling:**
  - **Non-pre-emptive** schemes include:
    - **The First Come, First Served (FCFS); similar to FIFO:**
      - In this scheme, which ever processes arrives to a ready state first, is run first.
      - This scheme does not take time that process needs into account, allowing slow jobs to be executed at any time causing performance degradation
      - This scheme also does not take priority into account. More important processes might wait for too long before being allowed to run.



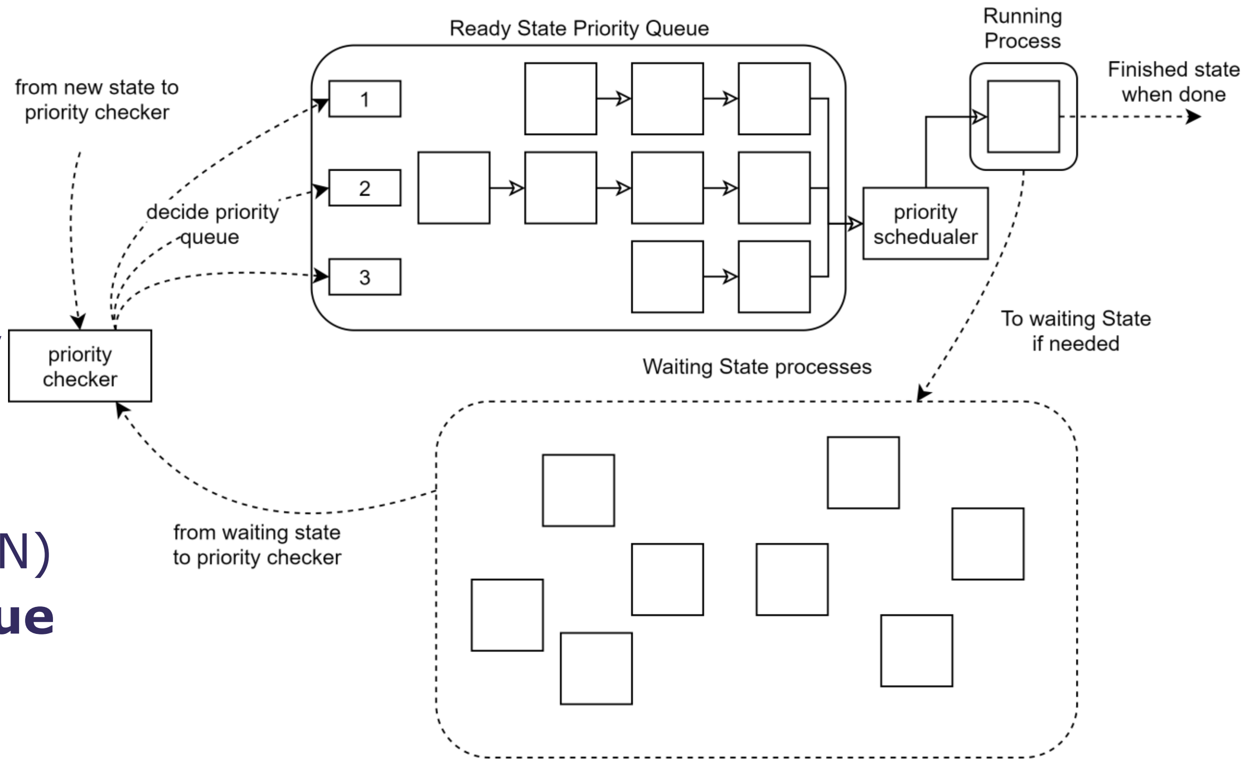
THE PROCESS

- **Process scheduling:**
  - **Non-pre-emptive** schemes include:
    - **Shortest Job Next (SJN):**
      - In this scheme, once a process comes to a ready state, the process scheduler makes a **run-time estimate** for it.
      - Jobs with the shortest run-time is added in head of the queue.
      - Shorter processes are ran first, allowing less wait time for shorter processes. A long process usually needs to be in waiting state multiple times before finishing.
      - This can still cause processes with high Priority to be in ready state for too long.
      - This can still cause resource hogging.



THE PROCESS

- **Process scheduling:**
  - **Non-pre-emptive** schemes include:
    - **Priority scheduling:**
      - In this scheme, processes are assigned to a priority queue. It works similar to SJN, but higher priority processes are ran first.
      - This priority can be decided using different information stored in the **PCB.**
      - This can include:
        - Process' mode: kernel / user
        - Process' user: OS / sys admin / priority user / user
        - Process run-time estimation: remaining time / needed time (SJN)
      - **Resource hogging is still an issue**



THE PROCESS

- **Process scheduling:**
  - **What if during running of a process, we decide that another process should run?**
    - Using the **non-pre-emptive** schemes, we need to wait for a process to finish.
    - But sometimes, we need to stop a running process to run a different process
    - For that we use a **pre-emptive** scheme.
    - **Pre-emptive** schemes: Where a process can be moved back to the ready state from the running state – **a process in running state can be stopped when needed. Once an I/O or a resource is needed: it is moved to the Waiting state.**
      - This helps avoiding **resource hogging** by monitoring each process and stopping it if it tries to take too many resources.
        - Those schemes are very **complex** to implement
        - They require **additional HW and SW for managing the PCBs** of the currently running and waiting processes.
      - These schemes can sometimes cause an overhead when used – additional delays happen because of the complexity.
      - In addition to the complexity, those schemes themselves can be a process. This means that the CPU will have to do extra work for those schemes to be effective.

THE PROCESS

- **Process scheduling:**
  - **Pre-emptive** schemes include:
    - **Shortest Remaining Time (SRT):**
      - It is the pre-emptive version of **SJN**. In this scheme, the process scheduler keeps track of the estimated run-time of processes in the Ready state and the estimated remaining run-time of a process in the Run state.
      - Consider the following scenario:
        - **Process A is in Ready state** it has 0.0033 seconds of estimated run-time.
        - **Process B is in Run state** it has 0.0024 seconds of remaining run-time.
        - **Process C is in New State** and it has 0.0012 seconds of estimated run-time.
      - What happens when **Process C** comes to ready state?

THE PROCESS

- **Process scheduling:**
  - **Pre-emptive** schemes include:
    - **Shortest Remaining Time (SRT):**
      - It is the pre-emptive version of **SJN**.
      - Consider the following scenario:
        - **Process A is in Ready state** it has 0.0033 seconds of estimated run-time.
        - **Process B is in Run state** it has 0.0024 seconds of remaining run-time.
        - **Process C is in New State** and it has 0.0012 seconds of estimated run-time.
      - What happens when **Process C** comes to ready state?
        - Since **SRT** favours processes with the shortest run time, **Process C is moved to the head of the Ready queue.**
        - Now, a process in the **Ready state** has estimated run-time shorter than the remaining run-time of the current process in the run state.
        - The process scheduler stops **Process B** and puts **Process C** in running state.
        - Since **process B** has less time that **Process A**, it is moved to the head of the Ready queue.

THE PROCESS

- **Process scheduling:**
  - **Pre-emptive** schemes include:
    - **Round Robin:**
      - In this scheme, processes are given a slice of CPU timing. This slice is called a Process **Time Quantum**.
      - Instead of forcing a Process to move from Running state to a Ready state, Round Robin decides a time quantum equal for each process.
      - Once this time quantum expires, the process is then moved to the tail of the Ready queue, allowing the process in the head of the queue to run.
      - **Time Quantum** allocations are extremely important for performance.
      - If a Process requires less time quantum than it has, its time quantum will finish once it finishes.
      - This requires an additional data structure to keep track of all this information.
        - This data structure is called a **Request Queue**.
        - A process is entered to the queue at the beginning of the access cycle.
      - Let us see an example of how this works:

ms

THE PROCESS

- **Round Robin:** In this example, the time quantum is **3 ms.**

Process Schedule		
P	Ready	Estimated RT
A	0 ms	3 ms
B	1 ms	7 ms
C	4 ms	2 ms
D	7 ms	5 ms

RT: Run-Time

CPU Run Time																			
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Request Queue	
P	Remaining RT

THE PROCESS

- **Round Robin:** In this example, the time quantum is **3 ms.**

Process Schedule		
P	Ready	Estimated RT
A	0 ms	3 ms
B	1 ms	7 ms
C	4 ms	2 ms
D	7 ms	5 ms

RT: Run-Time

CPU Run Time																			
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Request Queue	
P	Remaining RT
A	3

- **Process A** arrives to the Ready Queue and entered to the Request Queue.

THE PROCESS

- **Round Robin:** In this example, the time quantum is **3 ms.**

Process Schedule		
P	Ready	Estimated RT
A	0 ms	3 ms
B	1 ms	7 ms
C	4 ms	2 ms
D	7 ms	5 ms

RT: Run-Time

CPU Run Time																			
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Request Queue	
P	Remaining RT
A	3

- **Process A** arrives to the Ready Queue and entered to the Request Queue.
- Since there is no other Process in the request queue, it is set to Running state and receives a time quantum of 3 ms.
- **While A was running, at 1 ms, Process B also arrived to the Ready state and put into the request queue**

THE PROCESS

- **Round Robin:** In this example, the time quantum is **3 ms.**

Process Schedule		
P	Ready	Estimated RT
A	0 ms	3 ms
B	1 ms	7 ms
C	4 ms	2 ms
D	7 ms	5 ms

RT: Run-Time

CPU Run Time																			
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
A																			

Request Queue	
P	Remaining RT
~~A~~	~~3~~
B	7

- **Process A** only needed 3 ms to finish, once it is done, it is removed from the queue.
- **Process B** is sent to the running state, and given 3 ms quantum.
- **At 4 ms, Process C arrived to the Ready queue and put in the request queue**

THE PROCESS

- **Round Robin:** In this example, the time quantum is **3 ms.**

Process Schedule		
P	Ready	Estimated RT
A	0 ms	3 ms
B	1 ms	7 ms
C	4 ms	2 ms
D	7 ms	5 ms

RT: Run-Time

CPU Run Time																			
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
	A																		
				B															

Request Queue	
P	Remaining RT
~~A~~	~~3~~
~~B~~	~~7~~
C	2
B	4

- **Process B** used its quantum and ran for 3 ms seconds.
  - **It is then removed from the head of the queue and added to the tail of the queue with its remaining RT**
- **Next in the queue is process C.** it is given a 3 ms quantum, but it only needs 2 ms.
- **At 7 ms Process D arrives to Ready state. It is put into the request Queue.**

THE PROCESS

- **Round Robin:** In this example, the time quantum is **3 ms.**

Process Schedule		
P	Ready	Estimated RT
A	0 ms	3 ms
B	1 ms	7 ms
C	4 ms	2 ms
D	7 ms	5 ms

RT: Run-Time

CPU Run Time																			
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
	A																		
				B															
						C													

Request Queue	
P	Remaining RT
~~A~~	~~3~~
~~B~~	~~7~~
~~C~~	~~2~~
B	4
D	5

- **Process B** used its quantum and ran for 3 ms seconds again, but it did not finish yet.
  - **It is then removed from the head of the queue and added to the tail of the queue with its remaining RT**
- **Next in the queue is process D.** it is given a 3 ms quantum.

THE PROCESS

- **Round Robin:** In this example, the time quantum is **3 ms.**

Process Schedule		
P	Ready	Estimated RT
A	0 ms	3 ms
B	1 ms	7 ms
C	4 ms	2 ms
D	7 ms	5 ms

RT: Run-Time

CPU Run Time																			
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
	A																		
				B															
						C													
									B										

Request Queue	
P	Remaining RT
~~A~~	~~3~~
~~B~~	~~7~~
~~C~~	~~2~~
~~B~~	~~4~~
D	5
B	1

- **Process B** used its quantum and ran for 3 ms seconds again, but it did not finish yet.
  - **It is then removed from the head of the queue and added to the tail of the queue with its remaining RT**
- **Next in the queue is process D.** it is given a 3 ms quantum.

THE PROCESS

- **Round Robin:** In this example, the time quantum is **3 ms.**

Process Schedule		
P	Ready	Estimated RT
A	0 ms	3 ms
B	1 ms	7 ms
C	4 ms	2 ms
D	7 ms	5 ms

RT: Run-Time

CPU Run Time																			
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
	A																		
				B															
						C													
									B										
												D							

Request Queue	
P	Remaining RT
~~A~~	~~3~~
~~B~~	~~7~~
~~C~~	~~2~~
~~B~~	~~4~~
~~D~~	~~5~~
B	1
D	2

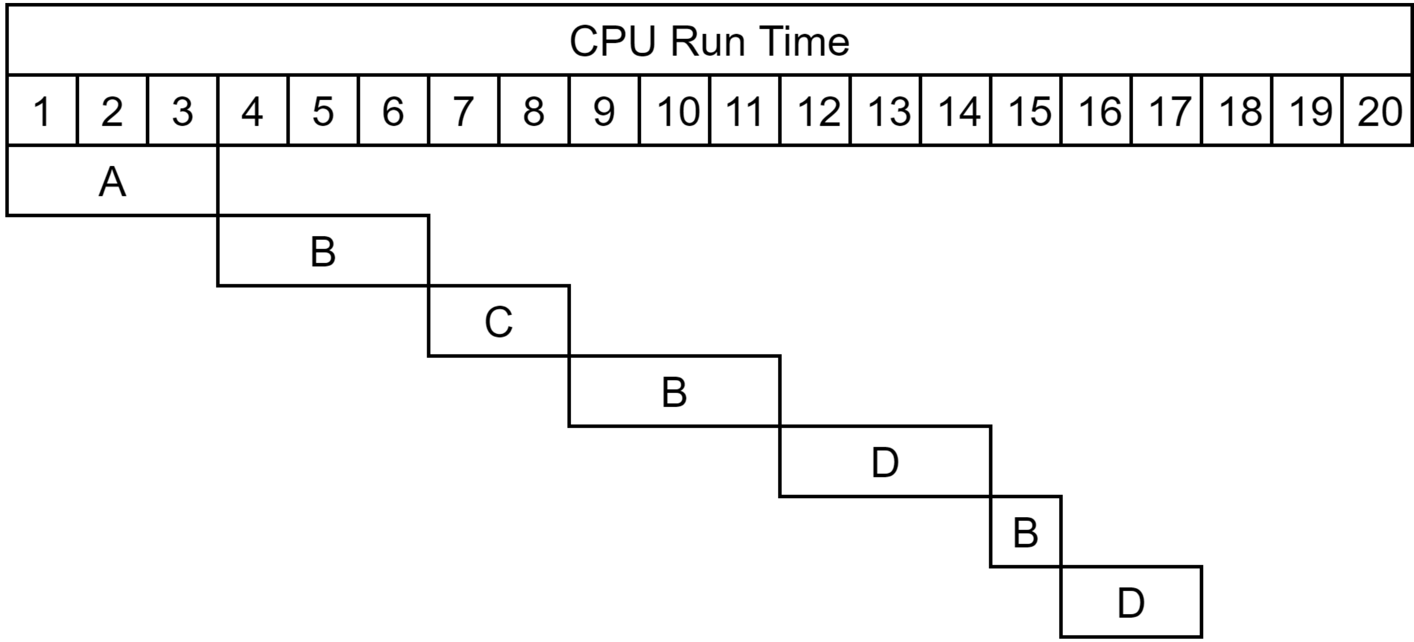
- **Process D** used its quantum and ran for 3 ms seconds again, but it did not finish yet.
  - **It is then removed from the head of the queue and added to the tail of the queue with its remaining RT**
- **Next in the queue is process B.** it is given a 3 ms quantum, but it only needs 1 ms, so it runs until if finishes, then **process D** is ran for 2 ms until it finishes

THE PROCESS

- **Round Robin:** In this example, the time quantum is **3 ms.**

Process Schedule		
P	Ready	Estimated RT
A	0 ms	3 ms
B	1 ms	7 ms
C	4 ms	2 ms
D	7 ms	5 ms

RT: Run-Time



Request Queue	
P	Remaining RT
~~A~~	~~3~~
~~B~~	~~7~~
~~C~~	~~2~~
~~B~~	~~4~~
~~D~~	~~5~~
~~B~~	~~1~~
~~D~~	~~2~~

All process in the request queue are done.

THE PROCESS

- **Context Switching:**
  - Now we know what a process is, what are the process states and **how it is changed from Ready state to Running.**
  - But we still do not know **how** it is changed from **Running state to Waiting/Ready state**.
  - **Context switching** is the mechanism **that a process is changed from the Running State to the Ready or Waiting state.**
  - Context switching happens when a Running state is interrupted because of an **event**.
  - An event is called an **interrupt**.
    - There are many types of interrupt:
      - Software interrupts: caused by software (program instructions)
        - Syscalls / Device Drivers / Sub-routines
      - Hardware interrupts
        - I/O inputs / Device Controllers / Pre-emptive process schedulers

THE PROCESS

- **Context Switching:**
  - Once an interrupt occurs, the CPU diverges its attention to handle an interrupt - if it is a software interrupt - or service the interrupt Request - if it was hardware interrupt.
    - Almost all types of interrupts are handled by the OS services.
    - Some execution signals coming from the peripherals of the CPU are handled by the CPU itself.
  - An interrupt can send a process into a Waiting state if it requires the process to wait for I/O or resources.
  - An interrupt can send a process into a Ready state if it is issued by the process scheduler