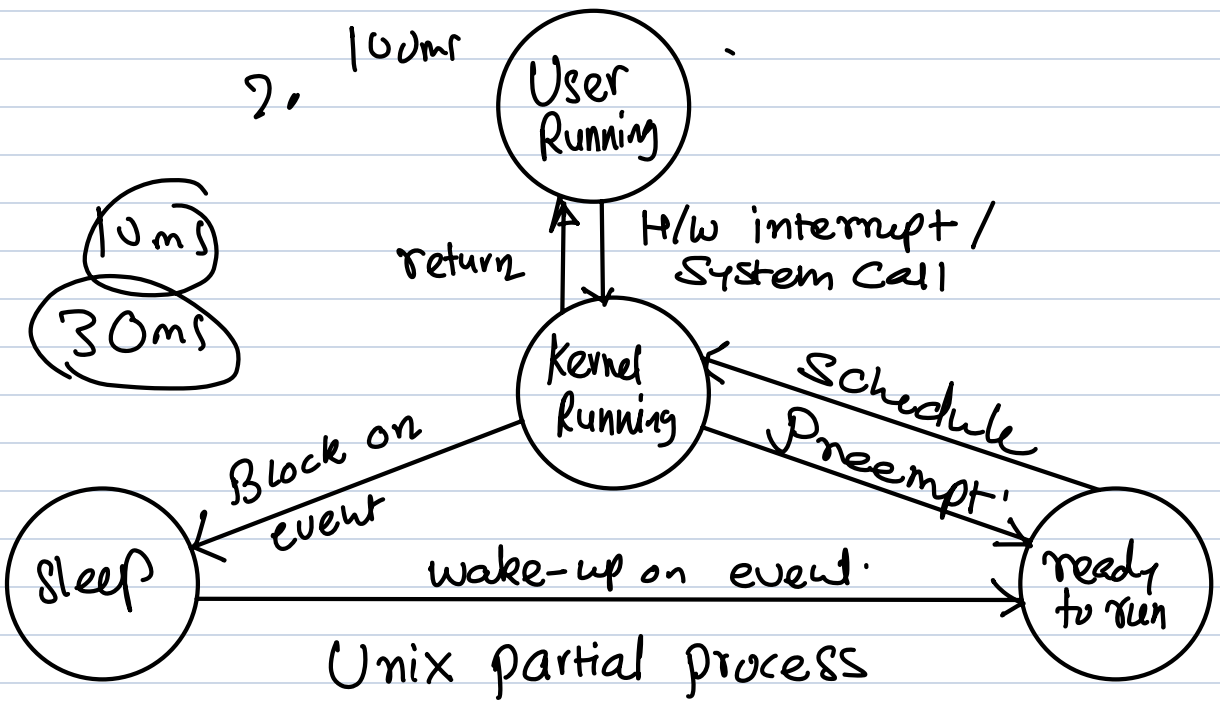
1) Single Tasking OS:

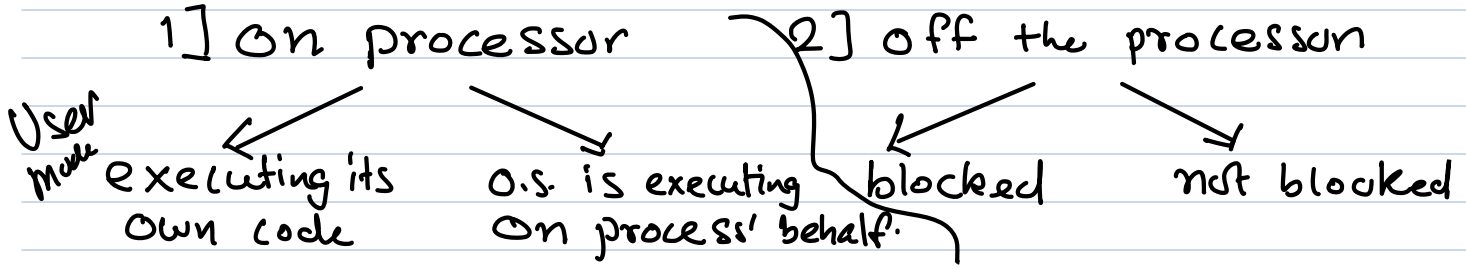
PC Power-On -> Single tasking OS boot (disk -> ram self transfer) -> Offer a command prompt -> User had entered the command -> The OS system used to load the appropriate executable file in RAM for execution -> Once the control flow is transferred to the application OS had to wait until termination of the application for regaining the control flow. -> Consider a scenario -> Where launched application goes into an inifinite loop then the end user had no choice but to restart the computer (or to send the interrupt to say the least)

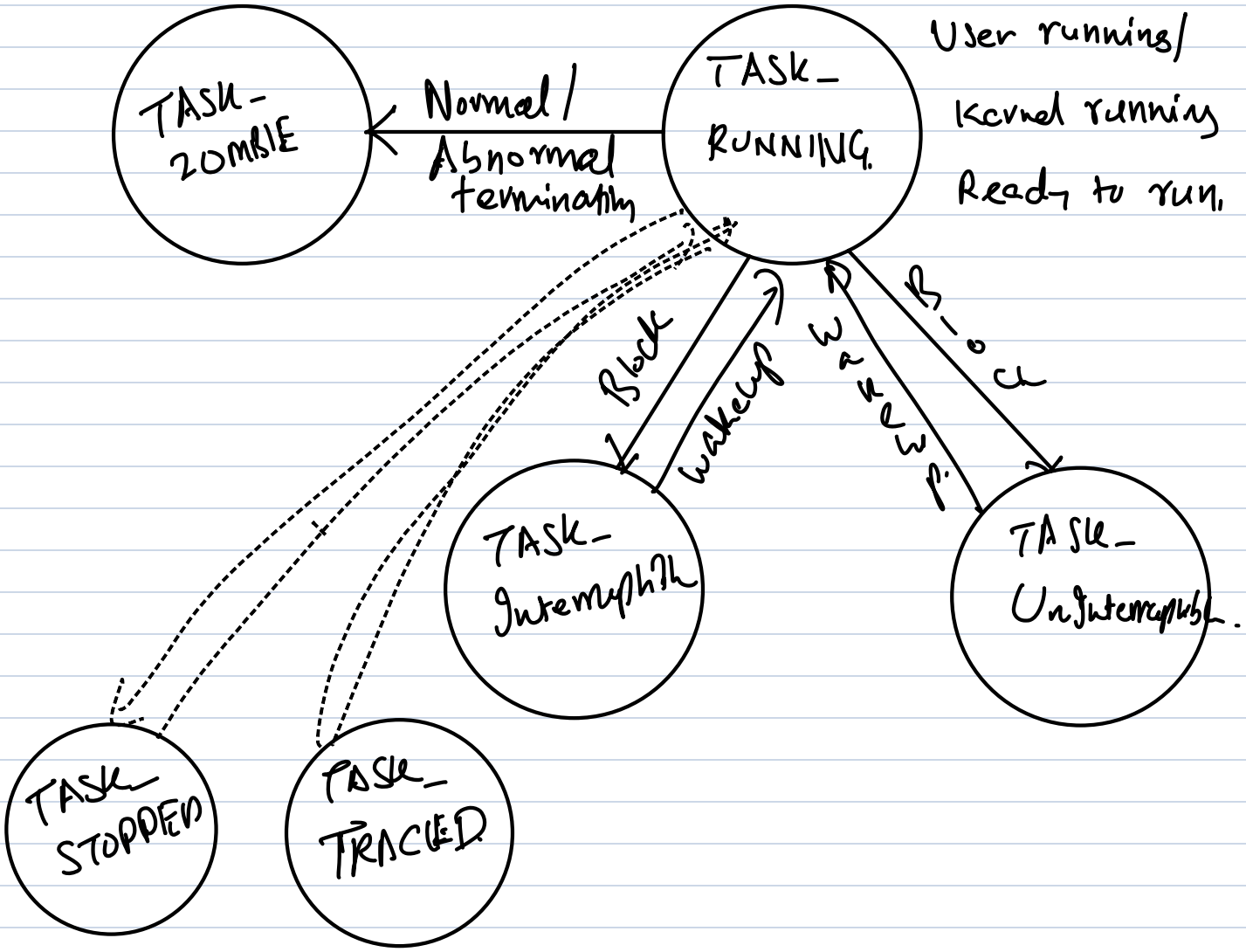
2) Multi-tasking OS: The end user could load more than one task into the main memory for the sake of execution. After single tasking operating system batch mode multi-tasking operating systems were invented. Though batch mode multi tasking operating systems allowed the end user to load more than one taks for the execution, the task themselves were executed sequentially (or in so called batch mode). Therefore, if the currently running task blocks for some reason then the freed CPU bandwidth couldn't be utilized by the rest of the tasks that had been loaded for the execution. This is a drawback of the batch mode multi-tasking.

3) Multi-tasking OS : Time sharing based multi-tasking operating system. -> Multiple tasks can be loaded in main memory for execution -> The operating system has a specialized component known as a schedular which has a privilege to determine which task should get a slice of microprocessor time -> Even though schedular selects a task for the execution and therefore the CPU is alotted to that task, it is not guaranteed that the task will finish in the given time duration and the CPU is not alotted on 'until done' basis. -> The OS typically saves the state of such task and suspends it and alotts CPU to rest of the tasks as well ensuring fairness.



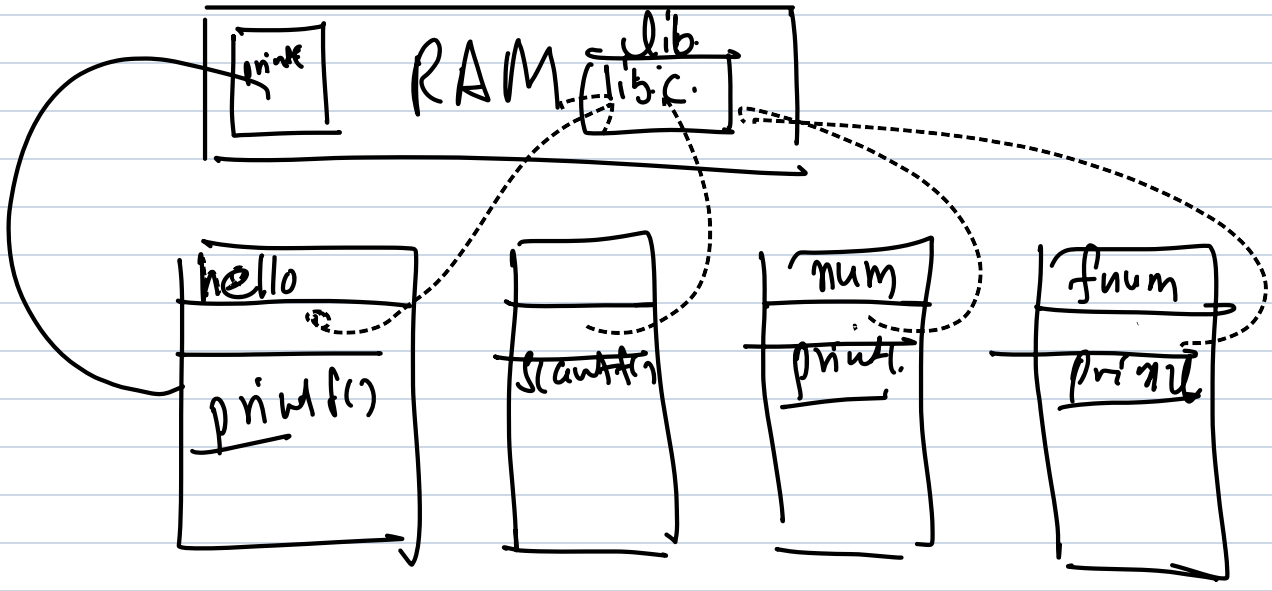
Unix partial process State diagram

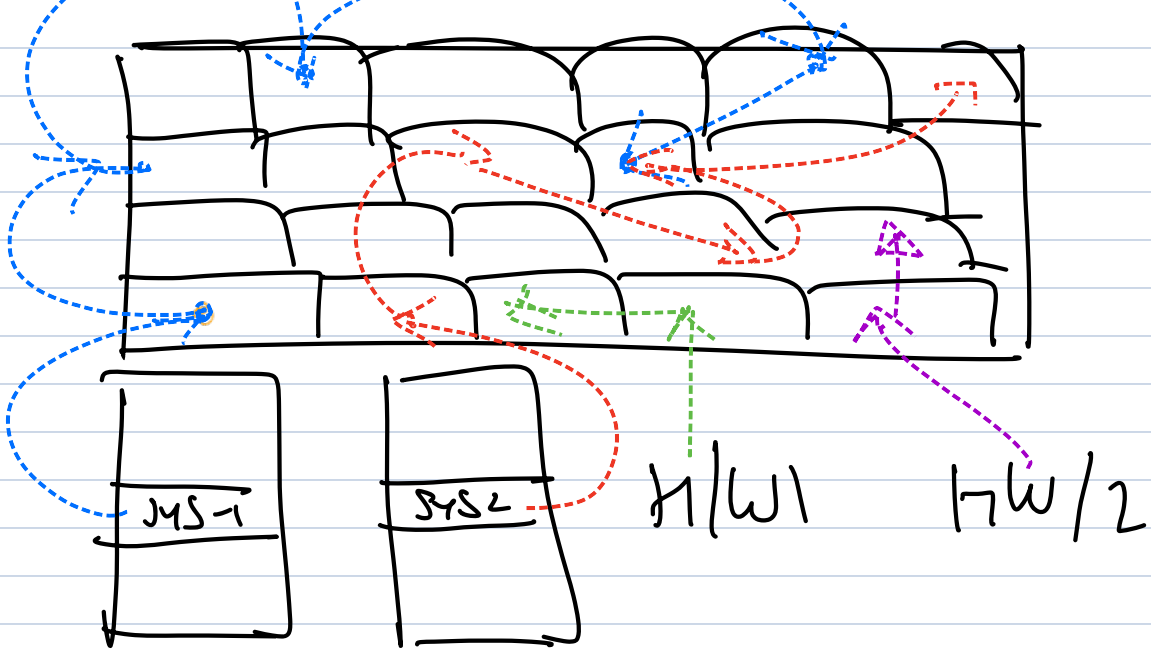




---

kernel control path (KCP).




SYS-1
SYS2
H/W1
H/W2