1.

READY to RUN: The process is allocated the CPU by the scheduler

READY to SUSPEND: No more memory is available, so the READY process is temporarily swapped out of memory

READY to BLOCKED: Not possible

RUN to READY: Time allocated to the particular process expires

RUN to BLOCKED: I/O request of the process is fulfilled

RUN to SUSPEND: Not possible

BLOCKED to READY: The awaited event completes (e.g. I/O completion)

BLOCKED to RUN: Not possible

SWAPPED to READY or SWAPPED to BLOCKED: When memory becomes available or the reason for swapping is no longer true

2.

T = 22: P5, P8 - ready/running

P1, P3, P7 - blocked for I/O

T = 37: P1, P3, P8 - ready/running

P5 - blocked suspend (swapped out)

P7 - blocked for I/O

T = 47: P1, P3, P5 - ready/running

P7 - blocked for I/O

P8 - exit

3. Fork return child process id if it is executed successfully else 0.

4. Mode Switch in threads is cheaper as compared to mode switch in process because control does not get transferred to Kernel. Also memory is shared between threads so there's no need to exchange memory or data during thread creation or switching.

5.

* Kernel is unaware about the ULTs, they can run on any OS without changing anything in the codes.
* Scheduling can be application specific.
* Since all of the thread management structures are within the user address space of a single process, the process does not switch to the kernel mode to do thread management. Thus saving time and no mode switching needed.

6

* When a ULT executes a system call, not only is that thread blocked, but also all of the threads within the process are blocked.
* In a pure ULT strategy, a multithreaded application cannot take advantage of multiprocessing. A kernel assigns one process to only one processor at a time. Therefore, only a single thread within a process can execute at a time.

7. When a thread issues an I/O request, kernel has to intervene to adhere to that request. This request comes to kernel in the name of process to which that thread belongs. Thus the kernel blocks the process which has issued the I/O request. It further results in blocking of threads running inside that process.

8. The program spends most of its time waiting for I/O operation to complete because it issues blocking system call. In a multithreaded program, one KLT can make the blocking system call, while the other KLTs can continue to run. On a uniprocessor machine, a process that would otherwise have to block for all these calls can continue to run its other threads.

9. No

10. Competing Process: Compete for resources and don't share/cooperate. This may lead to a race condition which OS needs to regulate

Cooperating Process: These processes may or may not be aware of each other but they share resources. They work jointly by sharing od resources by cooperating

11. Strong Semaphores uses FIFO approach to unblock processes using ***semSignal()*** while weak semaphore imposes no such constraint.

12. Monitors are programming language construct that provides equivalent functionality to that of semaphores and is easier to control.

13. Blocking send or receive will block the sender or receiver respectively until the message is received while non-blocking send or receive will not block the sender or receiver respectively. They are mainly used as listed here. Block send - blocking receive, Non-Blocking send - Blocking receive, Non-Blocking send - Non-Blocking receive:

14. No. It can be more efficient in the case where the expected waiting time is shorter than the time it takes to pre-empt and re-schedule a thread.

15. Yes. Functionally both codes are similar. It’s just that the semaphore code returns the value of number of processes blocked and is waiting in the queue to be executed.

16.

Loop

{

semwait(santa);

if(all\_reindeer\_ready)

{

for all\_waiting\_reindeer

{

}

for all\_ reindeer

{

}

Deliver Tops;

for all\_reindeer

{

}

}

Else if(all\_elves\_ready)

{

for all\_waiting\_elves

{

}

for all\_ reindeer

{

}

consult;

for all\_reindeer

{

}

}

}