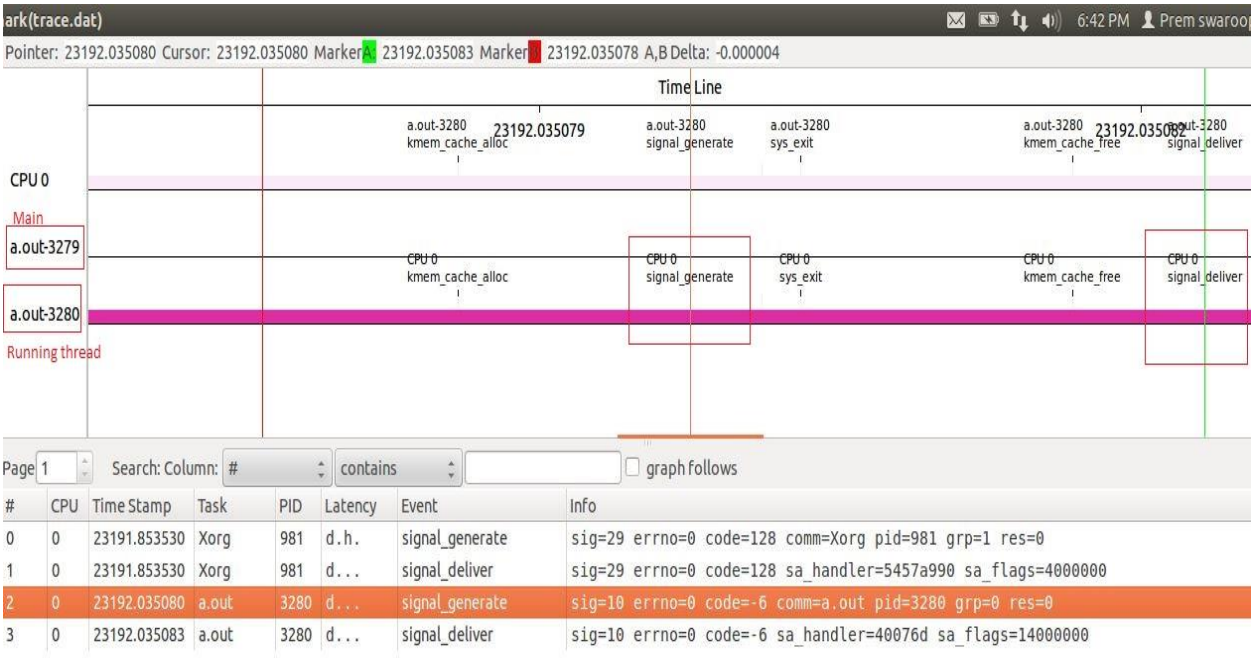
Signal Delivery Mechanism in Linux

The Description of signal delivery is operating system dependent. The following the analysis of signal delivery in various cases / states of threads will explain the mechanism in Linux.

a. Thread is running

When signal is delivered to thread when it is running its signal handler is executed in the same time. The graph from kernel shark below explains this.

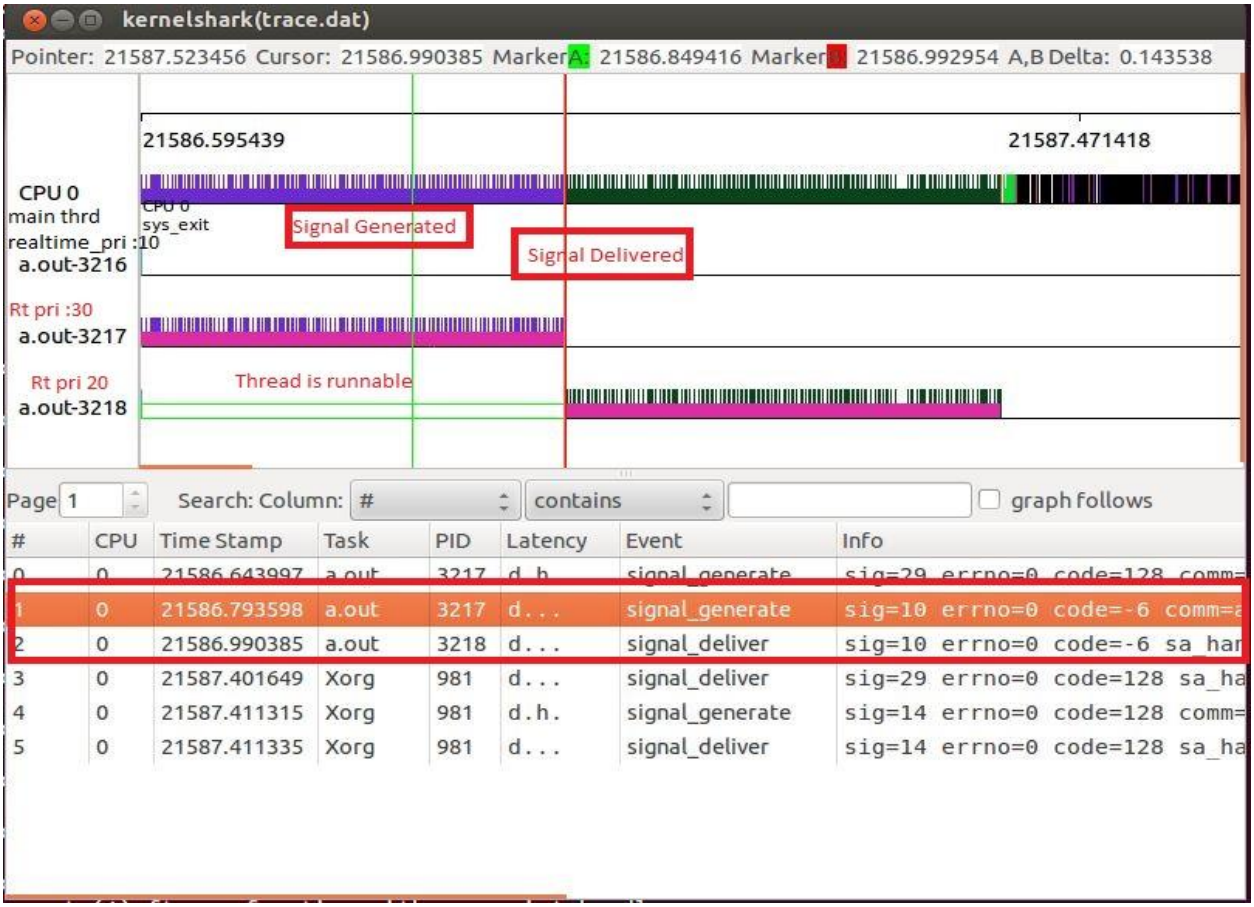


The task with id 3280 is thread created by main and signal is generated inside the thread so that is delivered when it is running. From the above graph it is observed that SIG#10 (SIGUSR1) is generated and delivered without getting preempted.

b. Thread is runnable

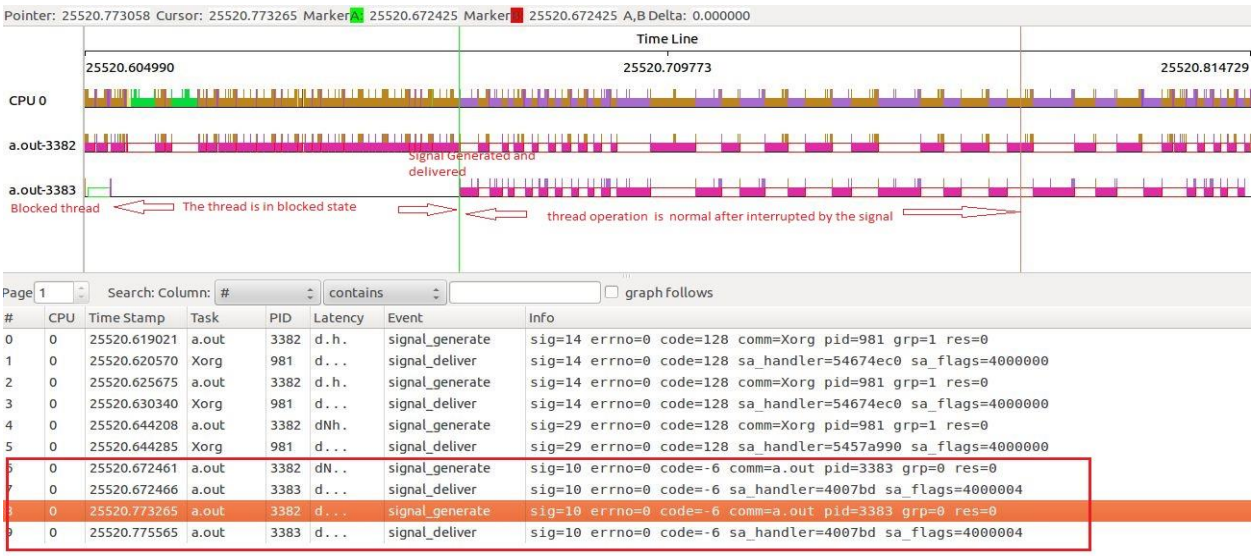
Thread is **runnable** i.e... When thread is ready to run but preempted by high priority thread then the signal is delivered after the thread is scheduled. The signal is queued and stored in pending signal set for that thread.

The below graph from kernel shark is obtained by creating 2 real-time threads with priorities 20 and 30. The high priority thread is scheduled first and generated the signal for low priority thread. The thread is not scheduled because of high priority thread is still running. When it completes its execution then the low priority signal is scheduled and the pending signal is delivered the taks2.

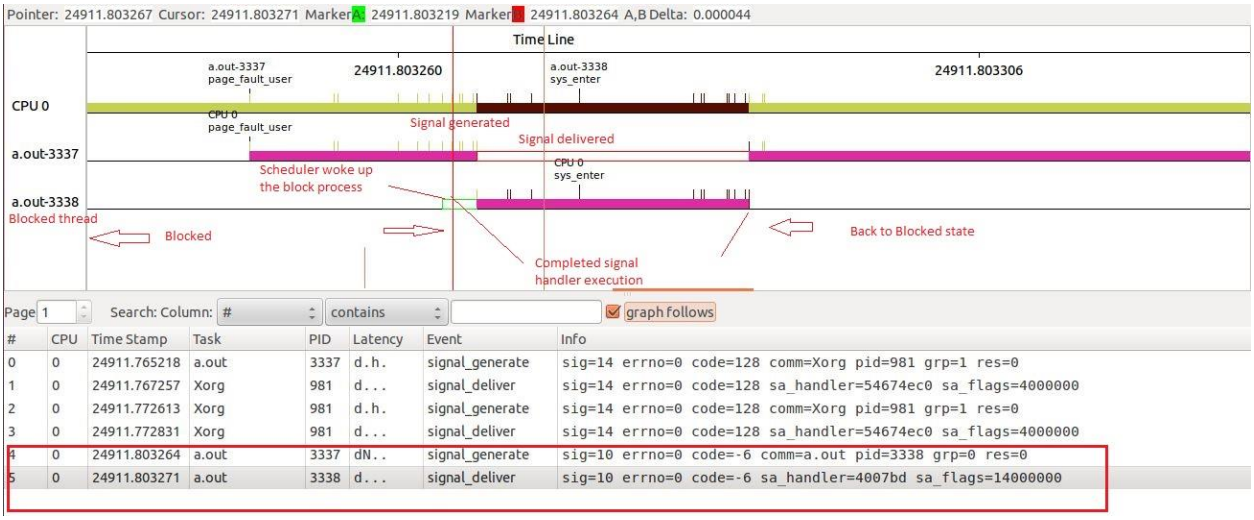


c. Thread is blocked

Thread is blocked on semaphore for a resource it is removed from the scheduler and kept as blocked. When a signal is delivered the scheduler wakes up the thread and delivers the signal to the thread.



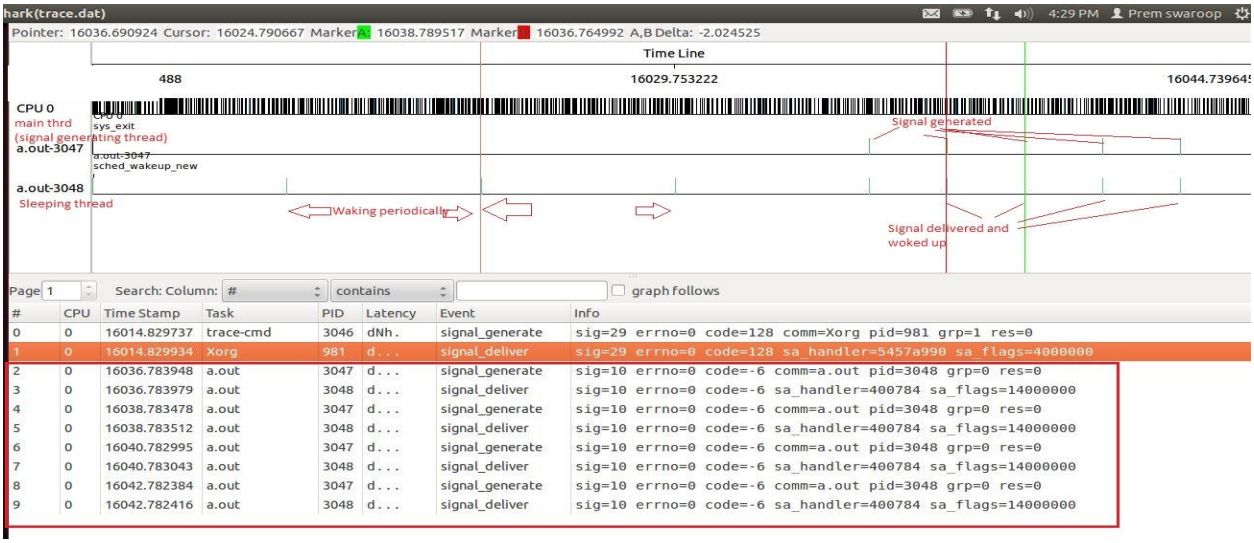
The call the **sem_wait** can be interrupted by the signal hence it is no more in blocked state after it is interrupted by the thread. The above graph for kernel shark explains this behavior. When we use the signal handler with SA_RESTART. The signal does not change the thread state. After waking up and executing the blocked thread it will again go back to the blocked state. The below graph explains this behavior.



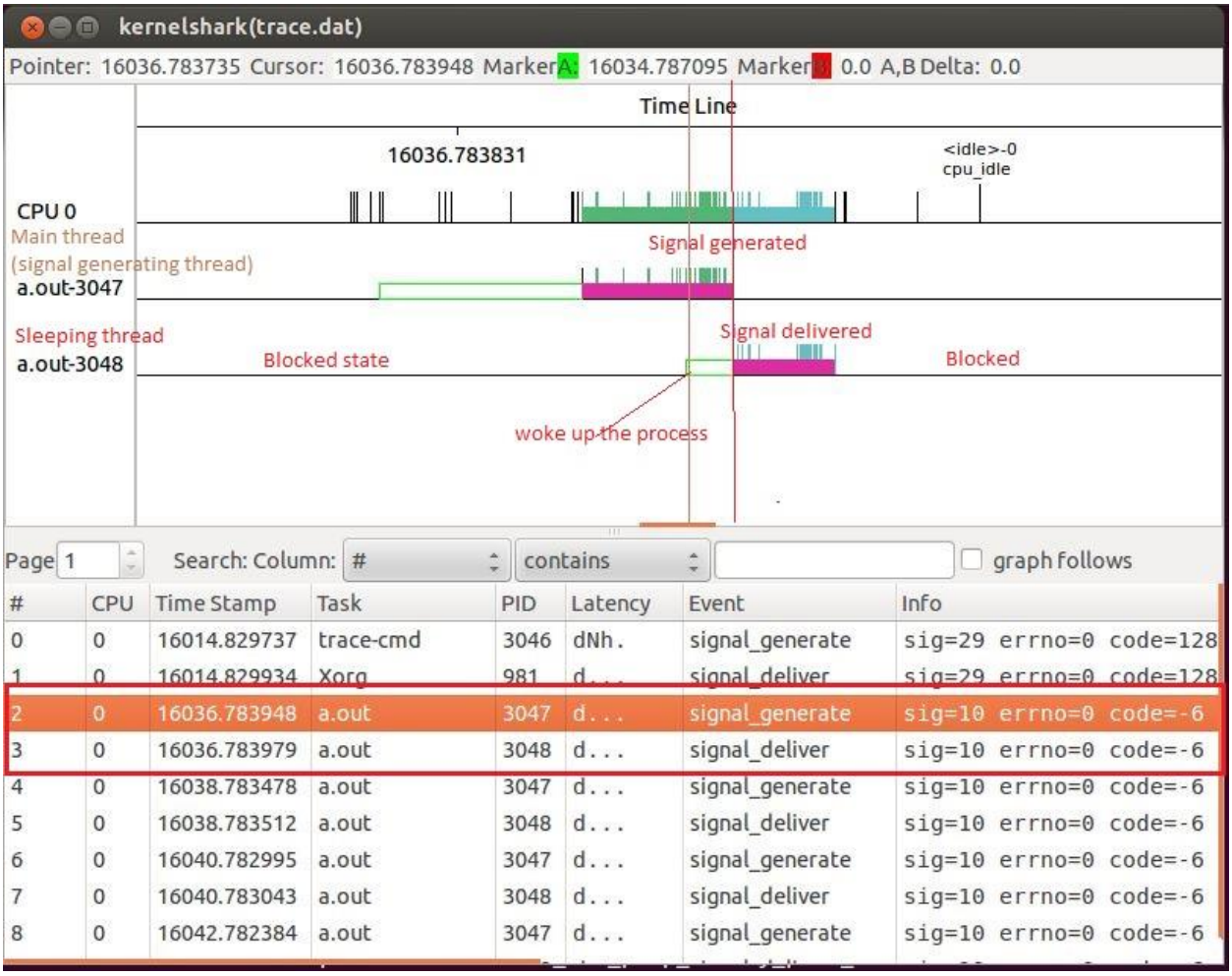
The hollow green bar indicates it is waked up and ready to execute (runnable) as soon as the other thread finishes its turn. After executing the signal handler the thread again resumed to its blocked state.

d. Thread is delayed

When thread is sleeping it is treated as delayed state. This can be observed from the below kernel shark graph. Initially the thread wakes up every 5 secs and executes and goes to sleep again. When signal arrived the thread is removed from delayed state and waked up scheduler and made it ready to run.



When it got chance it executes the signal handler and goes back to sleep. Hence before the signal arrived its period is observed as 5 and after threads start arriving it is interrupted by the threads arrival time. If the graph is zoomed in and observed closely the when signal is generated/raised the thread is interrupted from the delayed state and made it ready to run. It only executes the signal handler and it resumes back to sleep because it has nothing to but to sleep. The sleep function call is interruptible. It breaks from its sleep when signal is received. Hence we are continuously running sleep in thread it is rescheduled to wake up after its sleep interval. But as the signal interval is less the sleep interval it is interrupted by the signal handler. The zoomed in graph is shown below.



#	CPU	Time Stamp	Task	PID	Latency	Event	Info
0	0	16014.829737	trace-cmd	3046	dNh.	signal_generate	sig=29 errno=0 code=128
1	0	16014.829934	Xorg	981	d...	signal_deliver	sig=29 errno=0 code=128
2	0	16036.783948	a.out	3047	d...	signal_generate	sig=10 errno=0 code=-6
3	0	16036.783979	a.out	3048	d...	signal_deliver	sig=10 errno=0 code=-6
4	0	16038.783478	a.out	3047	d...	signal_generate	sig=10 errno=0 code=-6
5	0	16038.783512	a.out	3048	d...	signal_deliver	sig=10 errno=0 code=-6
6	0	16040.782995	a.out	3047	d...	signal_generate	sig=10 errno=0 code=-6
7	0	16040.783043	a.out	3048	d...	signal_deliver	sig=10 errno=0 code=-6
8	0	16042.782384	a.out	3047	d...	signal_generate	sig=10 errno=0 code=-6