CSE 438

REPORT

Event Handling and Signaling

Samruddhi Joshi
1213364722
Varun Joshi
1212953337

Problem Statement : To understand the signal delivery in various scenarios as below

In vxWorks' Kernel API Reference Manual, it is stated that "If a task is pended (for instance, by waiting for a semaphore to become available) and a signal is sent to the task for which the task has a handler installed, then the handler will run before the semaphore is taken. When the handler returns, the task will go back to being pended (waiting for the semaphore). If there was a timeout used for the pending task, then the original value will be used again when the task returns from the signal handler and goes back to being pended. If the handler alters the execution path, via a call to longjmp( ) for example, and does not return then the task does not go back to being pended."

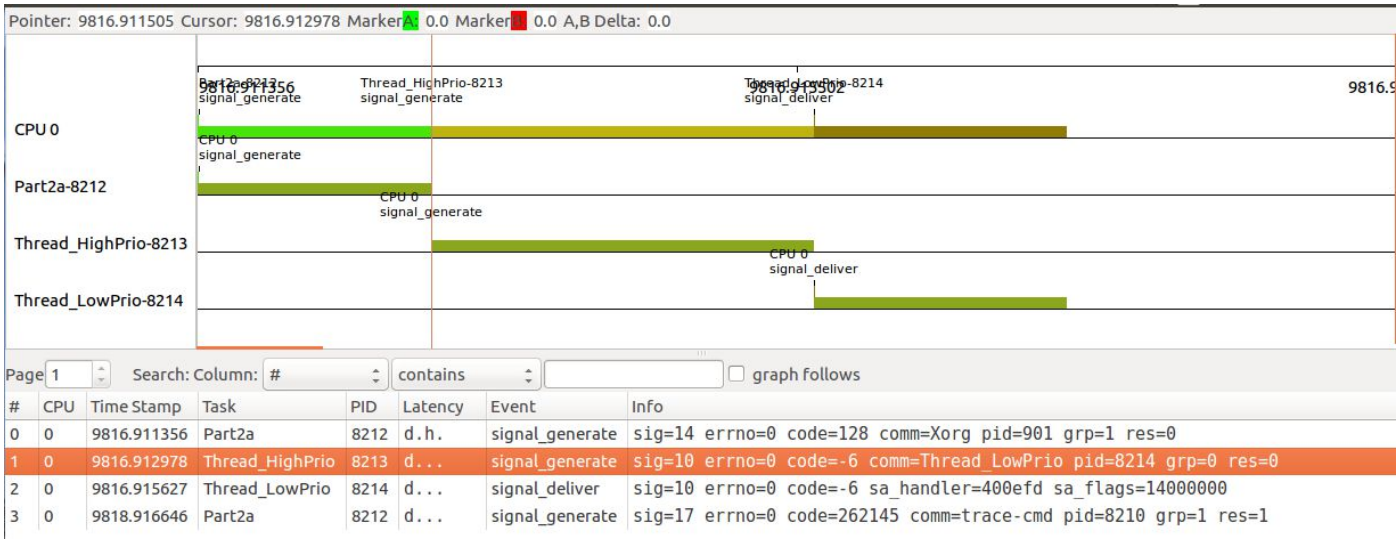
Show Linux's signal facility does and the time that a signal handler associated to a thread gets executed in the following conditions:

a. The thread is runnable (but not running, i.e. the running thread has a higher priority).

b. The thread is blocked by a semaphore (i.e. sema_wait() is called).

c. The thread is delayed (i.e., nanosleep() is called).

Part A : Thread is runnable (Higher priority thread is running)



#	CPU	Time Stamp	Task	PID	Latency	Event	Info
0	0	9816.911356	Part2a	8212	d.h.	signal_generate	sig=14 errno=0 code=128 comm=Xorg pid=901 grp=1 res=0
1	0	9816.912978	Thread_HighPrio	8213	d...	signal_generate	sig=10 errno=0 code=-6 comm=Thread_LowPrio pid=8214 grp=0 res=0
2	0	9816.915627	Thread_LowPrio	8214	d...	signal_deliver	sig=10 errno=0 code=-6 sa_handler=400efd sa_flags=14000000
3	0	9818.916646	Part2a	8212	d...	signal_generate	sig=17 errno=0 code=262145 comm=trace-cmd pid=8210 grp=1 res=1

Observation :

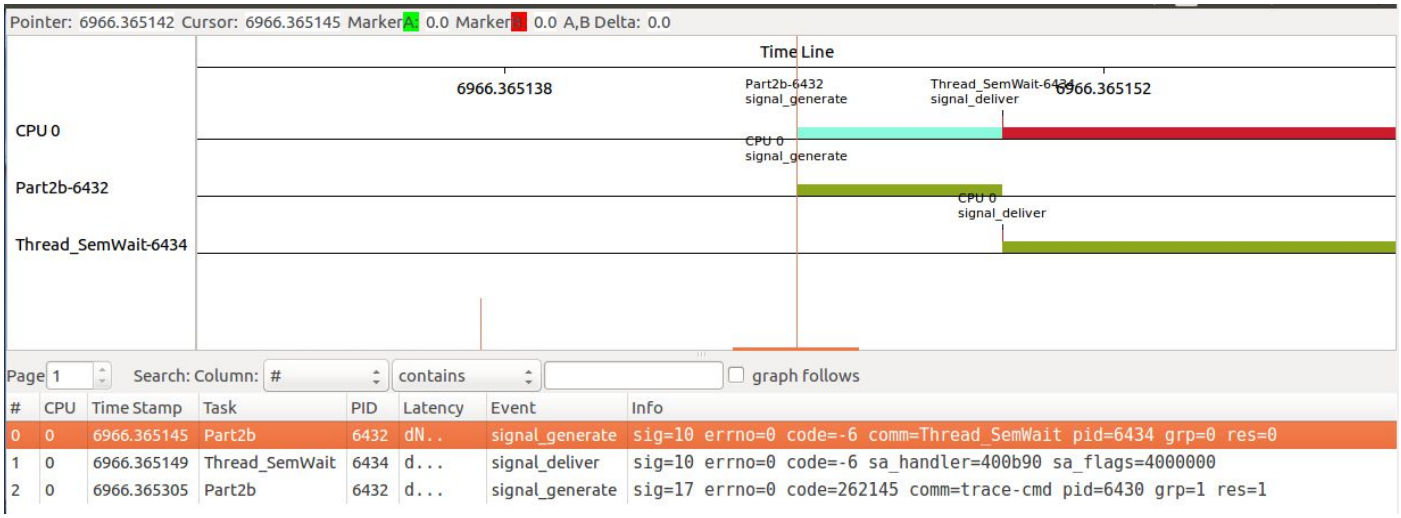
The High priority is running, the low priority thread is in runnable state. The High Priority thread generates signal to low priority thread which will be queued. **Signal will get delivered only when High Priority thread finishes its execution and then signal handler is called.**

Timestamp of signal_generate : 9816.912978

Timestamp of signal_deliver : 9816.915627

Difference : 2649 us

Part B : The thread is waiting for semaphore



Observation :

In this part,

1. Main thread is waiting for mouse event to occur
2. Thread_SemWait is blocked waiting for semaphore

As soon as 'Right Click' event occurs, the main thread generates a signal.

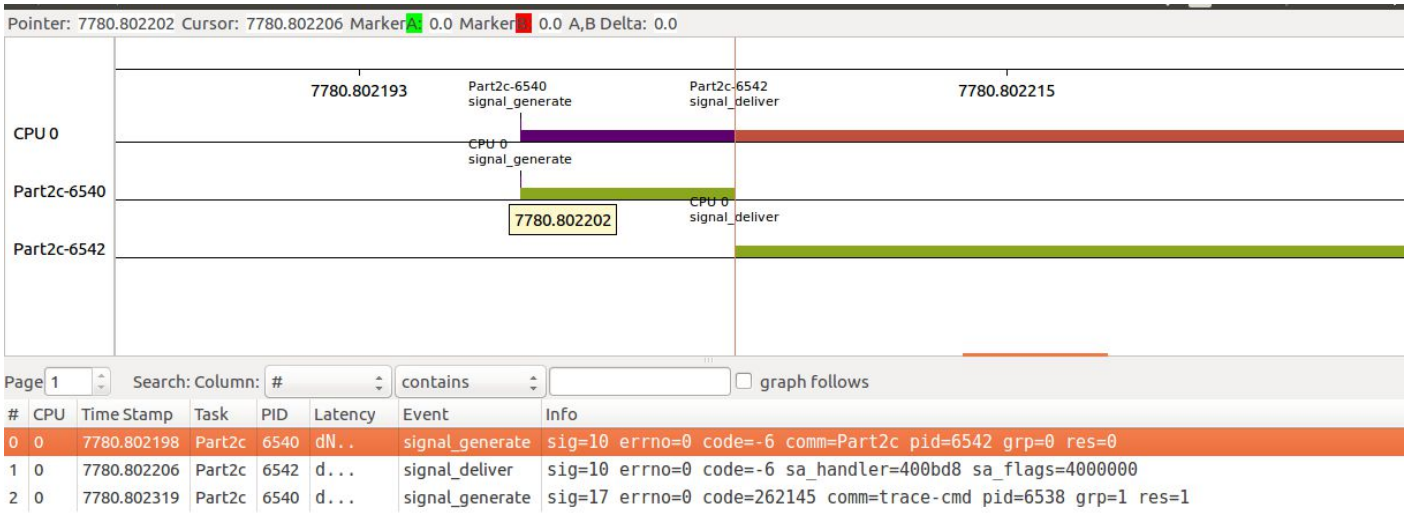
It is noted that the **signal handler is called even though the thread is in wait state.**

Once the control returns from the signal handler (even though no semaphore is posted) the Thread_SemWait is unblocked and starts executing.

It can been seen from the above snapshot, the timestamp difference between signal_generate and signal_deliver is only 4us.

We can say that If the semaphore currently has the value zero, then the call blocks until the semaphore becomes available, or a signal handler interrupts the call.

Part C: The thread is delayed due to nanosleep()



Observation :

In this part,

1. Main thread is waiting for mouse event to occur
2. Thread_Nanosleep is blocked by calling nanosleep()

If a 'Right Click' event occurs less than 10 secs from executing the binary, the main thread generates a signal. It is noted that the **signal handler is called even though the thread is in delayed state (nanosleep).**

Once the control returns from the signal handler, the Thread_Nanosleep wakes up even though the nanosleep period is not expired(it returns -1) and starts execution where it prints the remaining time.