IMPLEMENTATION DETAILS - AYUSH JAIN [jain207@purdue.edu]

Specific questions asked in lab4 for implementation are at the end

Steps taken for the implementation

a) A 2-D event table evtab[][] of type *evententry* was allocated of size (MAX_EVENTS+1) x (NPROC+1). Each element evtab[i][j] in this 2-D array corresponds to event no. i and process pid j. Since events are in the range of [1,64) and processes in range of [1,8], +1 has been added row and column size to reference events and processes by 1 indexing rather than 0 indexing.

b) The type of element in this 2-D belongs to struct type:

```
struct evententry{
        void (*event_handler)(uint32 event);
        int32 trigger_flag;
};
```

It contains pointer to the event handler function as well as the trigger_flag which is set once the event is triggered for a particular. There is no need to save pid and event no. as the index i,j in the 2-D refer to event no. i and process j as mentioned above.

c) Initialization of this 2-D array is done with default value of NULL and 0 for event_handler function pointer and trigger_flag in each evententry respectively.
regevent system call works by assigning the event_handler function pointer to the given event handler function pointer in the argument for a specific process and an event

d) **Regevent system call** works by checking the user event range in [32,64) & invalid pid and assigning the event handler function pointer in event entry(specific for the process calling regevent and event passed in the argument) to the given user function pointer in the argument.

e) This gives the user flexibility to update the function handler for a particular event by calling the regevent system call again. Also, the user can pass null as the value of function handler which can be assigned in that specific event entry implying that the user wants to unregister the event.

----

f) **Sendevent system call** checks for all the error conditions as mentioned in the lab requirements. After that, it sets the trigger_flag to 1 for the corresponding event and process pid which are passed as system call arguments.

g) In case an event of same type is sent multiple times while the event handler function has not been invoked specific to that process pid and event (i.e. trigger flag is already 1 for that event entry in event table), it returns SYSERR.

h) In case the pid which is being the sent the event is the current process running (i.e. process sending event to itself or some other process sending event to it while it is already scheduled), the event handler is executed immediately and the event trigger flag is set to 0 after execution.

i) In the other case when the process being sent event is not currently executing, the flag is used to run the event handler whenever the process resumes running.

j) The process resumes running right after ctxsw() call in resched as the context which is save right before rescheduling is restored before resuming the execution of process.

k) We check the trigger flag for that particular process across all the possible events in range [1,64) and see if the trigger flag is set for that particular event. In case the flag is set, the handler is invoked immediately followed by setting the event trigger flag to 0 and continues to check the flag corresponding to other events. Thus, it traverses an entire column in the 2-D array evtab.

Multiple events and event ordering

As mentioned above, the trigger flag is checked for a process across all possible events. This is done in ascending order in a for loop from 0 to 64 irrespective of which event is sent first. This is a disadvantage as events may be dependent on each other and need to be handled in the particular order as the events are sent. This will require a queue in that case but not considered as per lab requirement. Thus, no order is maintained for the events.

Handling processes with different priorities

m) This ensures that events are given normal priority i.e. a process that receives an event does not handle the event until the next time the scheduler selects the process for execution. This makes sense as the priority of event handling can be indirectly governed by the priority of the process. If the event requires a high priority, the event handler can be registered with the process with high priority.

n) This also ensures that the all types of event handlers will be invoked atleast once for which the event has been triggered while process was not running and flag was set for those events.

o) This approach has a disadvantage because if a process with high priority sends an event to a process with low priority, the event handler for the low priority process run may not run until the execution of high priority process is over which may introduce a large delay.

p) This also means that the process is “interrupted” in order to execute the event handler immediately if it receives event while it is running. The reason for implementation lies in the purpose of event handler in the first place. The registered event is to be handled as soon as the event is completed for that process. There is already a small delay between the time trigger flag is set and process is scheduled again. Introducing more delay can diminish the fast responsiveness of the event handler for the user.
For eg, if a button is clicked and process runs the handler after the execution of its existing code can be very frustrating for the user.

Event handlers and disabling/enabling interrupts

q) Running the event handler with interrupts enabled is the desired functionality as it is a user level function and operating system has higher privilege level than application. Thus, it may need to invoke software or hardware interrupts due to some error, hardware failure or other reason. In case the interrupts are disabled the system might crash.

r) For sending events in case of getmem and freemem system calls, the sendreservedevent is used which checks the event range in [1,32) and the rest of functionality is same as sendevent.

s) For removing the event handlers after termination of process, the event handler function pointer and trigger flag is set to -1 for that particular process across all events right before return of kill system call.

Extra Credit Implementation

t) Broadcast system call is implemented in a similar manner as send event system call . All the processes which have registered for a particular event is stored in an event table and their corresponding trigger flag is set to 1 in in case the process is not running or the handler is invoked immediately if process is the current process.

Specific questions asked in the lab

u) The test cases have been implemented in the following manner in code to check all the lab requirements:

Test case 1- Getmem and freemem event handling is checked for the current process.
Test case 2 - A single process receiving different type of events while one process is in waiting state
Test case 3 - Multiple processes registering different events and events being sent and handled to each process
Test case 9 - A process waiting for semaphore which has registered for 1 event handler. It receives multiple events of same type and handles only 1st event after resuming.
Test case 4 - A process registering events and sending events to itself

Test case 5 - Trying to send a reserved event resulting in error
Test case 6 - Trying to send a valid event who hasn't registered the handler for that event
Test case 7 (extra credit) - Process 2 and 4 registering for event 33 and sleeping for 2 and 10 sec respectively. Broadcast is called for event 33 handled by both the processes
Test case 8 - Existing system calls
Test case 10 - Removing event handlers and trigger flag after killing process

These test cases cover all the possibilities as per the lab requirements and considers them separately to check each functionality. This ensures that the code directly meets the specifications.

4 Operating System events that can be defined are as follows:

MEMORY_LIMIT_EVENT - In case the process is allocating memory such that it is almost reaching its threshold (say 8 bytes away from threshold) an event can be send to the user to handle this event by explicitly freeing some process memory.

TIMER_EVENT - An event can be sent after a specific number of clock ticks to the user for performing some function after a certain duration of time has elapsed.

MAX_PROCESS_LIMIT_EVENT - If a new process is created, an event can be sent if the number of active processes are reaching the max limit so that we can make sure to terminate some of useless processes that running.

DATA_AVAILABLE - In case data is available (say from a file into the the buffer), an event can be sent to the user to start making use of the data by consuming it while handling the event.