Problem 1 -- Process Scheduler (Simplified)

In this assignment, you will create an environment which simulates the issues of scheduling, forking, exiting and waiting for a child exit in a kernel. Although you may borrow ideas liberally from the Linux kernel, you will find that code is not directly transplantable, for the following reasons:

• Whereas the Linux kernel runs in supervisor mode and has complete control over all hardware, your routines will exist in a user-level testbed environment, running within one single-threaded UNIX process.

• The Linux kernel has the ability to create per-process virtual address space. You will not...all of your simulated tasks will exist within the single address space of the UNIX process. However, you will have to create and manage distinct user-level stacks for each task. Hints on doing this are given within.

• There will be no system call interface. Calls to your routines from the sample tasks will be via simple function calls.

• There are no hardware interrupts. You will simulate the periodic interval timer by means of UNIX signals.

• Your simulated scheduler will deal only with a simulated uniprocessor environment.

You must provide the following:

- A header file called `sched.h` which defines:

```
          struct sched_proc {
                    /* use this for each simulated task */
                    /* internals are up to you   */
                    /* probably should include things like the task state */
                    /* priority, accumulated, cpu time, stack address, etc. */
          };

          struct sched_waitq {
                    /* use this for each event/wakeup queue */
                    /* internals are up to you */
          };
          #define SCHED_NPROC XXX  // (maximum pid)-1, up to you but >=256
                    /* Suggested task state values, pick numbers */
          #define SCHED_READY     XXX
          #define SCHED_RUNNING   XXX
          #define SCHED_SLEEPING  XXX
          #define SCHED_ZOMBIE    XXX
```

- A file `sched.c` which implements the following functions:

```
sched_init(void (*init_fn)())
{
          /* This function will be called once by the testbed program.
                    It should initialize your scheduling system, including
                    setting up a periodic interval timer (see setitimer),
                    establishing sched_tick as the signal handler for
                    that timer, and creating the initial task which will
                    have pid of 1.  After doing so, make pid 1 runnable and
                    transfer execution to it (including switching to its
                    stack) at location init_fn.  This init_fn function is not
                    expected to return and if it does so it is OK to
                    have unpredictable results.
          */
}

sched_fork()
{
          /* Just like the real fork, create a new simulated task which
                    is a copy of the caller.  Allocate a new pid for the
                    child, and make the child runnable and eligible to
                    be scheduled.  sched_fork returns 0 to the child and
                    the child’s pid to the parent.  It is not required that
                    the relative order of parent vs child being scheduled
                    be defined.  On error, return -1.

                    Unlike the real fork, you do not need to duplicate the
                    entire address space.  Parent and child will execute in the
                    same address space.  However, you will need to create a
                    new private stack area for the child and initialize it to be
                    a copy of the parent’s.  See below for discussion on stacks.
          */
}

sched_exit(int code)
{
          /* Terminate the current task, making it a ZOMBIE, and store
                    the exit code.  If a parent is sleeping in sched_wait(),
                    wake it up and return the exit code to it.
                    There will be no equivalent of SIGCHLD.  sched_exit
                    will not return.  Another runnable process will be scheduled.
          */
}

sched_wait(int *exit_code)
{
          /* Return the exit code of a zombie child and free the
                    resources of that child.  If there is more
                    than one such child, the order in which the codes are
                    returned is not defined.  If there are no zombie children,
                    but the caller does have at least one child, place
                    the caller in SLEEPING, to be woken up when a child
                    calls sched_exit().   If there are no children, return
                    immediately with -1, otherwise the return value is the
                    pid of the child whose status is being returned.
                    Since there are no simulated signals, the exit code
                    is simply the integer from sched_exit().

          */
}

sched_nice(int niceval)
{
          /* Set the current task's "nice value" to the supplied parameter.
                    Nice values may range from +19 (least preferred static
                    priority) to -20 (most preferred).  Clamp any out-of-range
                    values to those limits
          */
}

sched_getpid()
{
          /* return current task's pid */
}

sched_getppid()
{
          /* return pid of the parent of current task */
}

sched_gettick()
{
          /* return the number of timer ticks since startup */
}

sched_ps()
{
          /* output to stderr a listing of all of the current tasks,
                    including sleeping and zombie tasks.  List the
                    following information in tabular form:
                              pid
                              ppid
                              current state
```

```
                                base addr of private stack area
                                static priority
                                dynamic priority info (see below)
                                total CPU time used (in ticks)
                 ** "dynamic priority" will vary in interpretation and range
                        depending on what scheduling algorithm you use.  E.g.
                        if you follow the CFS outline, then vruntime will be the
                        best indicator of dynamic priority
                */
                /* You should establish sched_ps() as the signal handler
                        for SIGABRT so that a ps can be forced at any
                        time by sending the testbed SIGABRT  */
        }



        sched_switch()
        {
                /* This is the suggested name of a required routine which will
                        never be called directly by the testbed. sched_switch()
                        should be the sole place where a context switch is made,
                        analogous to schedule() within the Linux kernel.
                        sched_switch() should place the current task on the run queue
                        (assuming it is READY), then select the best READY task from
                        the runqueue, taking into account the dynamic priority
                        of each task.  The selected task should then be placed
                        in the RUNNING state and a context switch made to it
                        (unless, of course, the best task is also the current task)
                        See discussion below on support routines for context switch.
                */
        }

        sched_tick()
        {
                /* This is the suggested name of a required routine which will
                        never be called directly by the testbed, but instead will
                        be the signal handler for the SIGVTALRM signal generated by
                        the periodic timer.  Each occurrence of the timer signal
                        is considered a tick.  The number of ticks since sched_init
                        is to be returned by sched_gettick().
                        sched_tick should examine the currently running
                        task and if its time slice has just expired, mark that
                        task as READY, place it on the run queue based on its
                        current dynamic priority, and then call sched_switch()
                        to cause a new task to be run.  Watch out for signal
                        mask issues...remember SIGVTALARM will, by default,
                        be masked on entry to your signal handler.
                /*
```

Your scheduling algorithm should have a range of static priority from 0 (best) to 39 (worst), with the default value of 20. You are free to imitate Linux in terms of dynamic priority and quantum time selection, e.g. by borrowing ideas from the O(1) scheduler or the CFS scheduler. You may also create any implementation of your choosing, consistent with the generic scheduling objectives of giving tasks which have a better static priority a greater share of the CPU.

Although in the real kernel, ticks come every 10ms or 1ms, it will be best if you slow things down and establish a tick rate of 100ms or so. Look up the setitimer system call.

SIGVTALRMs will therefore be arriving periodically. It is up to your sched_init() function to set this up and make sched_tick() the handler. Although you don't have to worry about multi-processor type synchronization issues, you do need to to consider what would happen if the SIGVTALRM signal arrives during one of your routines. Thus you need to protect critical regions by masking and unmasking these signals. This is analogous to a kernel for a uniprocessor system protecting critical regions by masking hardware interrupts.

It is suggested that you maintain a global variable containing either the current pid or the `struct sched_proc *` representing the current task.

Performing a context switch means saving the critical context information of the current task somewhere (in your case, the best place will be the `struct sched_proc` corresponding to the current task), and restoring that information for the switched-to task. Along with this assignment you have been supplied with the file `savectx.S` which defines two routines:

```
        int savectx(struct savectx *ctx);
        void restorectx(struct savectx *ctx,int retval);
```

They work similarly to setjmp/longjmp. In particular, `savectx` will return 0 when it is called directly, and `retval` when control returns to it via `restorectx`.

Because you are in a shared virtual address space, each simulated task needs its own stack area. Creating a new stack area is relatively easy in Linux:

```
void *newsp;
        if ((newsp=mmap(0,STACK_SIZE,PROT_READ|PROT_WRITE,
                MAP_PRIVATE|MAP_ANONYMOUS,0,0))==MAP_FAILED)
        {
                perror("mmap failed");
        }
```

You can define `STACK_SIZE` to be 64K or so and that will provide plenty of room for the testbed environment.

If you are switching from one existing task to another existing task, then at the point where you call `sched_switch()` in task A, your stack pointer is within the stack area that you have allocated for task A. Therefore, the SP which is saved by `savectx` will later be restored when task A is once again scheduled. However, how do you jump-start this process for creating the init process or forking a new process?

During a fork, you will allocate a new stack for the child, then `memcpy` the entire contents of the current (parent) stack into the new child stack. You know the base addresses of these two stacks, the difference between which (in bytes) is the adjustment which needs to be applied to stack-related addresses in the child stack. You will need to manipulate the `.regs[JB_SP]` and `.regs[JB_BP]` entries of the `struct savectx` context of the child so that when it is scheduled, the ebp and esp registers are correctly within the child's stack.

You'll also need to fix all the addresses in the predecessor functions to `sched_fork` in the child's stack. A support function `adjstack` has been supplied to you for that purpose. In the Linux kernel, this kluge is not necessary because each task has its own virtual address space. Since we are doing this all at user level, we do not have that tool at our disposal.

For establishing the init task, you could set up a `savectx` structure in which the program counter is the function representing the start of the init task, and the stack and base pointers are on the stack of the init task. You could then `restorectx`, thus jumping from the original invocation environment to the simulated multi-tasking environment.

You should have a global variable similar to the NEED_RESCHED flag in the Linux kernel. It may be set whenever a scheduler routine (e.g. sched_wakeup) decides that a context switch might be needed. It should be examined upon return from any signal handler (USR1,USR2 or VTALRM) and if set, sched_switch should be called to effect the context switch. Be careful when performing a context switch as you leave a signal handler. The signal mask will include the signal you have just handled. If that mask is not restored before the context switch, you may not receive that signal again!

You should create your own testbed programs to exercise your routines. A suggested test is to have your initial simulated task fork several times and create tasks with different nice priorities. Each task should call sched_exit and the parent task should call sched_wait. Use sched_ps to monitor the performance, e.g. call it whenever a context switch is being made. You should be able to demonstrate that your scheduler creates a fair allocation of CPU time among CPU-bound tasks and responds to "nice" values accordingly. It should be safe to send output to stdout or stderr from within your code at any time, although there is a possibility of truncated output if a signal arrives at a bad time.

Attach the output!!! (but please, if there is a lot of debugging output, trim the extraneous stuff!). Examine the performance of your code carefully to see that it implements static and dynamic priority. Watch out for resource leaks, endless loops, tasks which fail to get scheduled, and of course the inevitable unexpected exits and/or core dumps.