Assignment 1 Solutions

cpe 453 Spring 2010

```
It is equally bad when one speeds on the guest unwilling to go, and when he
holds back one who is hastening.  Rather one should befriend the guest who
is there, but speed him when he wishes.
-- Homer, "The Odyssey"

[Quoted in "VMS Internals and Data Structures", V4.4, when
 referring to scheduling.]
```

— /usr/games/fortune

Due by 11:59:59pm, Sunday, April 11th.
This assignment is to be done individually.

One of the principal tasks of an operating system is *scheduling*, that is, determining which process gets to run when. Your task for this assignment is to use your systems programming knowledge from csc/cpe357 to implement a simple user-level scheduler.

To do this, you will use your experience with both process management and signal handling.

Program: `schedule`

`Schedule` is a program that launches a series of child processes, then allows each to run for a given period (the *quantum*) before stopping it allowing the next process to run, etc., until there are no processes to be run.

Scheduling will be round robin, that is, each process takes its turn, in sequence, and the sequence is never changed, although processes can drop out by terminating.

The command line for the program looks like:

```
schedule quantum [prog1 [args] [: prog2 [args] [: prog3 [args] [: ...  ]]]]
```

The first argument, `quantum`, is the number of milliseconds a process is allowed to run before its turn is over. After that, come the command lines for each child, separated by colons[1]. The colon delimiting the command lines will be surrounded by space. That is, "`foo a : bar b`" is two commands, "`foo a`" and "`bar b`" while "`foo a: bar b`" is one command with arguments "`a: bar b`".

Because that's clear as mud, consider an example running three programs with a one-second quantum (`time(1)` shows how long it takes a program to complete, in this case, 7 hundredths of a second.):

```
% time schedule 1000 echo one : echo two : echo three
one
two
three
0.000u 0.002s 0:00.07 0.0%        0+0k 0+0io 0pf+0w
%
```

None of these programs gets a chance to be interrupted because there's not nearly enough work for a full second, but consider the following program, `two`. `Two` takes an integer argument, n, and

[1]Why colons? Why not? A single character that has no meaning to the shell seemed a good idea.

prints n, n times, in a field $8n$ characters wide, sleeping for one second in-between. Two runs are shown in Figure 1. In the first one, the quantum is large enough for each program to finish, but in the second, it is not.

```
% schedule 10000 two 1 : two 2 : two 3
       1
               2
               2
                       3
                       3
                       3
% schedule 500 two 1 : two 2 : two 3
       1
               2
                       3
               2
                       3
                       3
%
```

Figure 1: Effect on the schedule of changing the quantum

Your program must:

- `fork()` all of the processes in the order presented on the command line and stop them.
- Allow each process, starting with the first, to run until either its quantum expires, it terminates, or suspends itself.
- Use the interval timer (`setitimer()` with `ITIMER_REAL`) to keep track of when quanta have expired.
- Maintain the original round-robin scheduling order as processes terminate.
- Only give quanta to live processes. That is, if a process drops out, it should not be scheduled any more.
- Respond immediately to process terminations or self-suspensions. A process should not be forced to take its full quantum, and the next process should not have to wait for the quantum to expire.
- Give each process the opportunity to have a full quantum, regardless of how the previous process gave up its quantum.
- Continue scheduling processes until there are no more processes to schedule.
- `wait()` (or `waitpid()`) for each child at an appropriate time; you don't want to leave the system swarming with zombies.
- Not busy-wait. That is, it can't just sit and spin in a "`for(;;);`" loop waiting for signals. If you really don't have anything to do, `pause(2)`.

Even though it's not particularly difficult, for the purposes of this assignment, you need not consider the case where a stopped process terminates[2].

Tricks and Tools

There are many library routines and system calls that may help with implementing `schedule`. Some of them are listed in Figure 2.

`pid_t fork(2)`	creates a new process that is an exact image of the current one

`int execl(3)`
`int execlp(3)`
`int execle(3)`
`int execv(3)`
`int execve(2)`
`int execvp(3)`
	known collectively as "the execs", this family of functions overwrites the current process with a new program. For this assignment, look closely at `execvp()`.

`int kill(2)`	sends a signal to a process
`void pause(2)`	suspends execution until a signal is received.
`int raise(3)`	sends a signal to the current process. It is equivalent to `kill(getpid(), sig);`

`getitimer(2)`
`setitimer(2)`
	for getting and setting the system interval timer

`sigaction(2)`	the POSIX reliable interface for signal handling
`sigprocmask(2)`	for manipulating the blocking or unblocking of particular signals

`sigemptyset(3)`
`sigfillset(3)`
`sigaddset(3)`
`sigdelset(3)`
`sigismember(3)`
	for manipulating signal sets used by `sigaction(2)` and `sigprocmask(2)`

`strtol(3)`	convert a string to an integer

`pid_t wait(2)`
`pid_t`
`waitpid(2)`
	waits for a child process to change status (terminate or stop)

Figure 2: Some potentially useful system calls and library functions

In particular, pay attention to `waitpid()`. You have probably used `wait()` or `waitpid()` before to determine when a process has terminated, but when called with the `WUNTRACED` flag it will also return when a process has been stopped. Also, depending on your design, it could be if interest to you to know that whenever a process stops or terminates its parent receives a `SIGCHLD`.

Some hints:

- A process can be stopped by sending it a `SIGSTOP` signal, and a stopped process can be restarted with a `SIGCONT`.
- Remember that catching a signal interrupts "long" system calls like `wait()` and `pause()` and causes them to return.

[2]It is possible for this to happen if it receives an external signal.

- Remember to develop incrementally. Make sure you can launch and reap all of your children properly before attempting to schedule them.
- Think carefully before starting. This is not a large or complicated program, but there are a number of subtleties that it's best to discover before coding yourself into a corner.
- In particular, think about the problem of initally creating and stopping each child. The child process starts runnable, but you cannot allow it to run until the scheduler is ready for it to.

Coding Standards and Make

See the pages on coding standards and make on the cpe 453 class web page.

What to turn in

Submit via `handin` on the CSL machines to the `Asgn1` directory of the `pn-cs453` account:

- your well-documented source files.
- A makefile (called `Makefile`) that will build your program when run as just "`make`" or "`make schedule`".
- A README file that contains:
  - Your name, including your login name in parentheses (e.g. "(pnico)").
  - Any special instructions for running your program.
  - Any other thing you want me to know while I am grading it.

  The README file should be **plain text,** i.e, **not a Word document**, and should be named "README", all capitals with no extension.

Sample runs

There were some sample runs of `schedule` above. I will also place an executable version on the CSL in `~pn-cs453/demos` so you can run it yourself.

Solution:

File	**Where**
Makefile	p.5
debug.c	p.6
debug.h	p.7
sched.c	p.8
sched.h	p.12
timer.c	p.13
timer.h	p.15

```
CC     = gcc

CFLAGS  = -Wall -pedantic -g

PROG   = schedule

OBJS   = sched.o timer.o debug.o

SRCS   = sched.c

HDRS   = timer.h debug.h

EXTRACLEAN =

all:     $(PROG)

allclean: clean
        @rm −f $(EXTRACLEAN)

clean:
        rm −f $(OBJS) *~ TAGS

$(PROG): $(OBJS)
        $(CC) $(CFLAGS) −o $(PROG) $(OBJS)

depend:
        @echo Regenerating local dependencies.
        @makedepend $(SRCS) $(HDRS)

tags : $(SRCS) $(HDRS)
        etags $(SRCS) $(HDRS)

test:    $(PROG)
        $(PROG)
```

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/wait.h>
#include "debug.h"
#include "sched.h"


void report_on(pid_t proc, int status) {
  if ( proc == -1 ) {
    if ( errno == EINTR )
      fprintf(stderr, "wait() was interrupted\n");
    else
      perror("wait()");
  } else {
    if ( WIFEXITED(status)) {
      fprintf(stderr, "pid %d exited\n", proc);
    } else if ( WIFSIGNALED(status) ) {
      fprintf(stderr, "pid %d signalled\n", proc);
    } else if ( WIFSTOPPED(status) ) {
      fprintf(stderr, "pid %d stopped\n", proc);
    } else {
      fprintf(stderr, "pid %d did something...\n", proc);
    }
  }
}

void schedule_log(char *label, int old, int new, struct ptable *pt,
                  pid_t remove) {
  {
    int i;
    fprintf(stderr, "%s:  %s:  old=%d, new = %d, remove=%d, pt->count=%d, pt:  [",
            __FUNCTION__, label, old, new, remove, pt->count);
    for(i=0; i<pt->count; i++)
      fprintf(stderr, "%d%s", pt->table[i], (i!=pt->count-1)?", ":"");
    fprintf(stderr, " ]\n");
  }
}
```

```
#ifndef DEBUGH
#define DEBUGH

#include "sched.h"

void report_on(pid_t proc, int status);
void schedule_log(char *label, int old, int new, struct ptable *pt,
                  pid_t remove);

/* #define DEBUG */

#endif
```

```
#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/wait.h>
#include "sched.h"
#include "timer.h"


int main(int argc, char *argv[]) {
  struct ptable *ptable;
  int quantum;                  /* number of ms each process gets */
  char *endp;

  if ( argc < 2 ) {
    fprintf(stderr, "usage:  %s <quantum(ms)> p1 [ p1 args ] ; "
            "[ p2 [ p2 args ] ; [...]\n", argv[0]);
    exit(1);
  }

  quantum = strtol(argv[1],&endp,0);
  if ( !quantum || *endp != '\0') {
    fprintf(stderr, "%s is not a valid quantum\n", argv[1]);
    exit(2);
  }

  ptable = launch_procs(argc, argv);

  run_procs(ptable, quantum);

  if (ptable) {
    if (ptable->table)
      free(ptable->table);
    free(ptable);
  }

  return 0;
}


void start_process(pid_t which) {
  /* start the given process (with SIGCONT) */
  #ifdef DEBUG
  fprintf(stderr,"%s:  Starting process %d.\n",__FUNCTION__,which);
  #endif
  kill(which, SIGCONT);
}

void stop_process(pid_t which) {
  /* stop the given process (with SIGSTOP) */
  pid_t proc;
  int status;
  #ifdef DEBUG
  fprintf(stderr,"%s:  Stopping process %d.\n",__FUNCTION__,which);
  #endif
  if ( -1 != kill(which, SIGSTOP)) {
    /* wait for the child to stop */
    proc = waitpid(which, &status, WUNTRACED);
    if ( proc != which || !WIFSTOPPED(status) ) {
      fprintf(stderr, "%s:  process %d failed to stop\n", __FUNCTION__,
              which);
      exit(10);
    }
  }
}

pid_t launch_stopped(char *argv[]) {
  /* fork and exec a single process.  The child stops itself,
   * and the parent waits() for that before returning.
   */
  pid_t child, proc;
  int status;

  if ( !(child=fork()) ) {
    raise(SIGSTOP);             /* stop the child process */
    execvp(argv[0], argv);      /* proceed with the exec  */
    perror(argv[0]);
    exit(−1);
  } else {
    /* wait for the child to stop */
    proc = waitpid(child, &status, WUNTRACED);
    if ( proc != child || !WIFSTOPPED(status) ) {
      fprintf(stderr, "%s:  child failed to stop\n", __FUNCTION__);
      exit(10);
    }
  }

  return child;
}

struct ptable *launch_procs(int argc, char *argv[]) {
  /* parse the command line and start each process, suspended */
  struct ptable *pt;
  char **s, **t;
  int idx;

  pt = (struct ptable*)malloc(sizeof(struct ptable));
  if ( pt ) {
    /* count the processes in the table */
    pt−>count=1;
    for(s=argv+2; *s; s++)
      if (!strcmp(*s,DELIMITER))
        pt−>count++;

    /* allocate the table */
    pt−>table = (pid_t*) malloc(pt−>count * sizeof(pid_t));
    if ( ! pt−>table ) {
      perror("malloc");
      exit(3);
    }

    /* launch the processes */
    idx = 0;
    for(s=argv+2; *s; s=t) {
      /* find the end of this stage */
      for (t=s; *t && strcmp(*t,DELIMITER); t++)
        ;
      if ( *t )
        *t++ = NULL;
      pt−>table[idx++]=launch_stopped(s);
    }
  }
  return pt;
}

int schedule(int old, struct ptable *pt, pid_t remove) {
  /* returns the index of the next process to run out
   * of ptable.  If remove is nonzero, it represents
   * a process that has exited and should be removed
   * from the table.
   *   If no processes remain, returns 0;
   */
  int new, v, j;

  #ifdef DEBUG
  schedule_log("Before", old, new, pt, remove);
  #endif
  if ( !remove ) {
    /* nothing much to do.  Move along */
    new = (old+1) % pt−>count;
  } else {
    /* find the victim */
    for(v=0; pt->table[v] != remove; v++)
      ;

    /* move the table down */
    pt->count--;
    for(j=v; j < pt->count; j++)
      pt->table[j] = pt->table[j+1];

    /* now, schedule */
    if ( old < v ) {
      /* only advance the index if old < victim.  Otherwise
       * the table will move to it.  Also takes care of the
       * zero-count problem.
       */
      new = (old+1) % pt->count;
    } else {
      new = old;
    }
  }
  #ifdef DEBUG
  schedule_log("After", old, new, pt, remove);
  #endif
  return new;
}

void run_procs(struct ptable *ptable, int quantum) {
  /* Runs the processes in the given table, if any.
   * Things that can interrupt a wait():
   *   + the timer goes off
   *   + a running process stops on its own
   *   + a running process terminates
   *   + a non-running process terminates
   */
  int i, status;
  pid_t proc;
  if ( ptable && ptable->count) {

    /* run the processes */
    i=0;
    while(ptable->count) {
      /* start the ith process */
      start_process(ptable->table[i]);

      /* wait for something interesting to happen */
      start_timer(quantum);     /* start the timer */
      proc=waitpid(-1, &status, WUNTRACED);
      stop_timer();             /* stop the quantum ticker*/
      #ifdef DEBUG
      report_on(proc, status);
      #endif

      /* now, respond appropriately */
      if ( proc == -1 ) {
        /* If we were interrupted, the timer went off
         * or something bad happened
         */
        if ( errno == EINTR ) {
          /* the timer went off, stop this one and schedule  */
          stop_process(ptable->table[i]);
          i = schedule(i, ptable, 0);
        } else {
          perror("waitpid()");
          exit(5);
        }
      } else {
        /* it's an actual process that caused this.  Figure out
         * what happened and deal with it.
         */
        if ( WIFEXITED(status) || WIFSIGNALED(status) ) {
          /* A process terminated.  If it's not the running
```

```
             * one, stop the running one, then schedule.  This is
             * a little unfair, but, well, too bad.
             */
            if ( proc != ptable->table[i])
              stop_process(ptable->table[i]);
            i = schedule(i, ptable, proc);
          } else if ( WIFSTOPPED(status) ) {
            /* if it stopped, move on */
            i = schedule(i, ptable, 0);
          } else {
            /* If it neither stopped nor exited, I don't know what's
             * going on.
             */
            fprintf(stderr, "%s:  process status (0x%08x) neither exited "
                    "or stopped.    "
                    "That's puzzling\n", __FUNCTION__, status);
            exit(6);
          }
        }
      }
    }
}
```

```
#ifndef SCHEDH
#define SCHEDH

#define DELIMITER ":"

struct ptable {
  int count;
  pid_t *table;
};

struct ptable *launch_procs(int argc, char *argv[]);
void run_procs(struct ptable *ptable, int quantum);

#endif
```

```
#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/wait.h>
#include "debug.h"

typedef void (*sigfun)(int);

static void handler(int signum) {
  /* dummy handler.   We only want the signal to interrupt wait() */
  #ifdef DEBUG
  fprintf(stderr,"Ding!");
  #endif
}

void start_timer(int quantum) {
  /* install a signal handler for SIGALARM, then request one for
   * quantum milliseconds later
   */
  struct sigaction sa;
  struct itimerval it;
  int sec, usec;

  /* install the handler */
  sa.sa_handler=(sigfun)handler;
  sigemptyset(&sa.sa_mask);
  sa.sa_flags = 0;
  if ( −1 == sigaction(SIGALRM, &sa, NULL) ) {
    perror("sigaction");
    exit(7);
  }

  /* start the timer */
  sec = quantum/1000;
  usec = (quantum%1000) * 1000;
  it.it_interval.tv_sec  = 0;
  it.it_interval.tv_usec = 0;
  it.it_value.tv_sec  = sec;
  it.it_value.tv_usec = usec;
  if ( −1 == setitimer(ITIMER_REAL,&it,NULL) ) {
    perror("setitimer");
    exit(8);
  }
}

void stop_timer() {
  /* remove the signal handler and stop the interval timer */
  struct sigaction sa;
  struct itimerval it;

  /* remove the handler */
  sa.sa_handler=(sigfun)SIG_DFL;
  sigemptyset(&sa.sa_mask);
  sa.sa_flags = 0;
  if ( −1 == sigaction(SIGALRM, &sa, NULL) ) {
    perror("sigaction");
    exit(7);
  }

  /* start the timer */
  it.it_interval.tv_sec  = 0;
  it.it_interval.tv_usec = 0;
  it.it_value.tv_sec  = 0;
  it.it_value.tv_usec = 0;
  if ( −1 == setitimer(ITIMER_REAL,&it,NULL) ) {
    perror("setitimer");
    exit(8);
  }
}
```

timer.h

```
void start_timer(int quantum);
void stop_timer();
```