CIS 415 Project 1

This document discusses a model solution to Project 1.

General Discussion

The handout indicated that you needed to implement four versions of the Multiple Occurrence Test Harness, TH. You were given a set of low-level functions, defined in **p1fxns.h** and implemented in **p1fxns.c**, and forbidden to use any Linux chapter 3 functions ***EXCEPT*** **malloc()**, **free()**, **calloc()**, **realloc()**, **getenv()**, **execvp()**, **sysconf()**, and **getopt()**.

thv1

This version had to process the environment variables **TH_QUANTUM_MSEC**, **TH_NPROCESSES** and **TH_NCORES**; process the command line arguments (**./thv? [-q <msec>] [-n <nprocesses>] [-c <ncores>] –l 'cmdline'**), process the **cmdline** into an argument vector for **fork()**, create **nprocesses** child processes by forking a child process and execing the command in that child process, and then wait for each child process to terminate.

Preliminaries

The beginning of the source file **thv1.c** needs to perform the appropriate includes, define the sizes of some parameters, define the struct for a process control block, and define some global data. We also define two static functions, one to print the usage string for the program and exit, and the other to print the timing information specified in the handout. Note that **print_timing()** uses several features of **p1strpack()** in order to generate the output in the mandated format.

```
#include "p1fxns.h"
#include <unistd.h>   /* defines _exit() */
#include <stdlib.h>   /* defines NULL, getenv() */
#include <sys/wait.h> /* needed for wait() */
#include <sys/time.h>   /* needed for gettimeofday(), struct timeval */

#define MAXBUF 4096   /* maximum size of command + args */
#define MAXPCBS 512   /* maximum number of child processes */
#define MAXARGS 128   /* maximum number of arguments in each command */

typedef struct pcb {
    pid_t pid;        /* the pid associated with that process */
} PCB;

/*
 * global data
 */
PCB pcb_array[MAXPCBS];
int num_procs = 0;
int quantum = 0;
int nprocesses = 0;
int ncores = 0;
int active_processes = 0;
```

```
static void print_usage_and_exit(char *name) {
    p1putstr(2, "usage: ");
    p1putstr(2, name);
    p1putstr(2, " [-q <msecs>] [-n <nprocesses>] [-c <ncores>] -l 'command
line'\n");
    _exit(1);
}

static void print_timing(struct timeval *start, struct timeval *stop,
                         char *cmd, int nprocesses, int ncores) {
    char temp[256], tint[10], *sp;
    int msecs;

    msecs = stop->tv_usec / 1000 - start->tv_usec / 1000;
    msecs += 1000 * (stop->tv_sec - start->tv_sec);
    sp = temp;
    sp = p1strpack("The elapsed time to execute ", 0, ' ', sp);
    p1itoa(nprocesses, tint);
    sp = p1strpack(tint, 0, ' ', sp);
    sp = p1strpack(" copies of \"", 0, ' ', sp);
    sp = p1strpack(cmd, 0, ' ', sp);
    sp = p1strpack("\" on ", 0, ' ', sp);
    p1itoa(ncores, tint);
    sp = p1strpack(tint, 0, ' ', sp);
    sp = p1strpack(" cores is ", 0, ' ', sp);
    p1itoa(msecs/1000, tint);
    sp = p1strpack(tint, -3, ' ', sp);
    sp = p1strpack(".", 0, ' ', sp);
    p1itoa(msecs%1000, tint);
    sp = p1strpack(tint, -3, '0', sp);
    sp = p1strpack("sec\n", 0, ' ', sp);
    p1putstr(1, temp);
}
```

main()

The main function performs the following actions:

- Process the environment variables and arguments; make sure the user has supplied valid **quantum**, **nprocesses**, and **ncores** values, and has specified the **cmdline** to execute in the child processes.
- Parse the **cmdline** into an argument vector for use in ***execvp()***.
- Note the start time.
- Create **nprocesses** child processes by calling **fork()**; in the child branch of the test after calling **fork()**, ***execvp()*** the **cmdline**; in the parent branch, store the pid of the child process in the pcb_array.
- Wait for all child processes to terminate.
- Note the end time, and print the elapsed time as instructed in the handout.
- Clean up after ourselves and exit.

Gather and validate global information

```
int main(int argc, char **argv) {
    char cmd[MAXBUF];
    char *sp = NULL;
    char *progname = *argv;
    int n, opt;
    pid_t pid;
    int i, j;
```

```
    char word[MAXBUF];
    PCB *p = pcb_array;
    char *args[MAXARGS];
    struct timeval start, stop;

/*
 * process environment variables and command argument to th?
 */
    cmd[0] = '\0';
    if ((sp = getenv("TH_QUANTUM_MSEC")) != NULL)
        quantum = p1atoi(sp);
    if ((sp = getenv("TH_NPROCESSES")) != NULL)
        nprocesses = p1atoi(sp);
    if ((sp = getenv("TH_NCORES")) != NULL)
        ncores = p1atoi(sp);
    opterr = 0;
    while ((opt = getopt(argc, argv, "q:n:c:l:")) != -1) {
        switch(opt) {
        case 'q': quantum = p1atoi(optarg); break;
        case 'n': nprocesses = p1atoi(optarg); break;
        case 'c': ncores = p1atoi(optarg); break;
        case 'l': p1strcpy(cmd, optarg); break;
        default:
            p1putstr(2, "illegal option: -");
            p1putchr(2, optopt);
            p1putstr(2, "\n");
            print_usage_and_exit(progname);
        }
    }
    if (quantum == 0) {      /* quantum must be specified */
        p1putstr(2, "quantum undefined\n");
        print_usage_and_exit(progname);
    }
    if (nprocesses == 0) {   /* number of processes must be specified */
        p1putstr(2, "number of processes undefined\n");
        print_usage_and_exit(progname);
    }
    if (ncores == 0) {       /* number of cores must be specified */
        p1putstr(2, "number of cores undefined\n");
        print_usage_and_exit(progname);
    }
    if (cmd[0] == '\0') {    /* command to execute must be specified */
        p1putstr(2, "command to execute undefined\n");
        print_usage_and_exit(progname);
    }
```

Convert **cmd** into argument vector, note start time, and create child processes

```
/*
 * at this point, we know:
 *    the command to execute in each process
 *    the number of processes to create running the command
 *    the number of cores over which the processes should be scheduled
 *
 * now we need to:
 *     parse the command string into program to execute and arguments
 *     fork `nprocesses' new processes to execute the command,
 *           storing its pid in the PCB
 */

    i = 0;
    j = 0;
    while ((i = p1getword(cmd, i, word)) != -1) {
        args[j++] = p1strdup(word);
```

```
        }
        args[j] = NULL;
        (void) gettimeofday(&start, NULL);
        for (n = 0; n < nprocesses; n++) {
            switch((pid = fork())) {
                case -1: p1putstr(2, "Error forking new process\n");
                          _exit(1);
                          break;
                case 0: /* child branch */
                         execvp(args[0], args);
                         p1putstr(2, "Child process: error execvp'ing ");
                         p1putstr(2, cmd);
                         p1putstr(2, "\n");
                         for (j = 0; args[j] != NULL; j++)
                             free(args[j]);
                         _exit(1);
                         break;
                default:/* parent branch */
                         p->pid = pid;
                         p++;
            }
            num_procs++;
        }
        active_processes = num_procs;
```

Wait for child processes to terminate, note end time, and print timing information

```
    /*
     * now wait for all child processes to terminate
     */
        while (active_processes > 0) {
            (void) wait(NULL);  /* wait for a child process to terminate */
            active_processes--;
        }
        (void)gettimeofday(&stop, NULL);
        print_timing(&start, &stop, cmd, nprocesses, ncores);
```

Cleanup and return

```
        for (j = 0; args[j] != NULL; j++)
            free(args[j]);
    /*
     * successful return
     */
        return EXIT_SUCCESS;
    }
```

thv2

Building on **thv1**, this version had to enable each child process to wait for a **USR1** signal before execing the command; the TH had to create all children, then send all children the **USR1** signal, then send all children a ***STOP*** signal, then send all children a ***CONT*** signal, before then waiting for all children to terminate.

There are very few changes needed to produce **thv2** from **thv1**. Rather than show the entire code for **thv2**, I show the output produced by executing the following command:

```
% diff --context=3 thv1.c thv2.c
```

I provide commentary within the diff output describing what it means.

```
*** thv1.c     2022-09-26 15:27:17.029287701 -0700
--- thv2.c     2022-09-26 15:27:22.508025945 -0700
***************
*** 3,9 ****
--- 3,12 ----
  #include <stdlib.h> /* defines NULL, getenv() */
  #include <sys/wait.h>     /* needed for wait() */
  #include <sys/time.h>   /* needed for gettimeofday(), struct timeval */
+ #include <time.h>       /* needed for nanosleep(), struct timespec */
+ #include <signal.h>     /* signal(), kill(), USR1, STOP, CONT */

+ #define UNUSED __attribute__((unused))
  #define MAXBUF 4096 /* maximum size of command + args */
  #define MAXPCBS 512 /* maximum number of child processes */
  #define MAXARGS 128 /* maximum number of arguments in each command */
***************
```

This indicates that **thv2** has three lines added to **thv1**, a **#include** for **<signal.h>** so the program can access **signal()**, **kill()**, and various ***SIG**** definitions, a **#include** for **<time.h>** so the program can access **nanosleep()** and the **timespec** structure definition, and a **#define** for **UNUSED** to avoid compiler warnings for our signal handler. The '+' character in the first column shows the lines that are added, with the three leading lines and the three trailing lines being the lines that are common in the two files.

```
*** 20,26 ****
  int quantum = 0;
  int nprocesses = 0;
  int ncores = 0;
! int active_processes = 0;

  static void print_usage_and_exit(char *name) {
      p1putstr(2, "usage: ");
--- 23,37 ----
  int quantum = 0;
  int nprocesses = 0;
  int ncores = 0;
! volatile int active_processes = 0;
! volatile int USR1_seen = 0;
!
! /*
!  * SIGUSR1 handler
!  */
! static void onusr1(UNUSED int sig) {
!     USR1_seen++;
! }

  static void print_usage_and_exit(char *name) {
      p1putstr(2, "usage: ");
***************
```

This indicates that the declaration for **active_processes** in **thv1** is replaced by several lines in **thv2**: **volatile** declarations for **active_processes** and for another global variable (**USR1_seen**), and the signal handler for **SIGUSR1**. The lines replaced in **thv1** are flagged with an exclamation mark in column 1, and the replacement lines in **thv2** are also flagged with an exclamation mark in column 1.

```
***************
*** 66,71 ****
--- 77,83 ----
      PCB *p = pcb_array;
      char *args[MAXARGS];
      struct timeval start, stop;
+     struct timespec ms20 = {0, 20000000};    /* 20 ms */
```

```
    /*
     * process environment variables and command argument to th?
  ***************
```

This indicates that **thv2** has an additional variable, **ms20**, which is a **struct timespec** that is initialized to 20ms.

```
  *** 108,113 ****
  --- 120,132 ----
              print_usage_and_exit(progname);
        }
    /*
  +  * establish USR1 signal handler
  +  */
  +     if (signal(SIGUSR1, onusr1) == SIG_ERR) {
  +         p1putstr(2, "Can't establish SIGUSR1 handler\n");
  +         _exit(1);
  +     }
  + /*
     * at this point, we know:
     *    the command to execute in each process
     *    the number of processes to create running the command
  ***************
```

This indicates that **thv2** has code added to establish the ***SIGUSR1*** signal handler. Note that child processes inherit signal handlers across a call to **fork()**.

```
  *** 132,137 ****
  --- 151,158 ----
                          _exit(1);
                          break;
                case 0: /* child branch */
  +                     while (! USR1_seen)
  +                         (void)nanosleep(&ms20, NULL);
                        execvp(args[0], args);
                        p1putstr(2, "Child process: error execvp'ing ");
                        p1putstr(2, cmd);
  ***************
```

This indicates that **thv2** has code added for the child process to wait for receipt of ***SIGUSR1***.

```
  *** 148,153 ****
  --- 169,189 ----
        }
        active_processes = num_procs;
    /*
  +  * now send USR1 signal to all child processes
  +  */
  +     for (i = 0; i < num_procs; i++)
  +         (void) kill(pcb_array[i].pid, SIGUSR1);
  + /*
  +  * now send STOP signal to all child processes
  +  */
  +     for (i = 0; i < num_procs; i++)
  +         (void) kill(pcb_array[i].pid, SIGSTOP);
  + /*
  +  * now send CONT signal to all child processes
  +  */
  +     for (i = 0; i < num_procs; i++)
  +         (void) kill(pcb_array[i].pid, SIGCONT);
  + /*
     * now wait for all child processes to terminate
     */
```

```
        while (active_processes > 0) {
```

This indicates that **thv2** has code added for the TH to first send **USR1** to all children, then send ***STOP*** to all children, and then send ***CONT*** to all children.

thv3

Building on **thv2**, this version had to enable an interval timer corresponding to the chosen quantum, properly implement round robin scheduling, properly reap terminated children, and gracefully terminate after all children have terminated.

In order to implement round robin scheduling, one needs to have a queue implementation that supports adding to the end of the queue and retrieving from the front of the queue. In the following, I have used the ArrayQueue from the Oregon ADT library.

thv3.c

It is easiest to show the differences against **thv1.c**, since many of the additions going from **thv1** to **thv2** are unused in **thv3**. Therefore, I show the output produced by executing the following command:

% diff --context=3 thv1.c thv3.c

rather than reproduce the entirety of the source code for **thv3**. I provide commentary within the diff output describing what it means; where some of the changes were documented when going from v1 to v2, I do not describe those changes again.

```
*** thv1.c      2022-10-03 10:45:17.530580733 -0700
--- thv3.c      2022-10-03 10:35:16.477240599 -0700
***************
*** 1,15 ****
--- 1,28 ----
  #include "p1fxns.h"
+ #include "ADTs/arrayqueue.h"
  #include <unistd.h> /* defines _exit() */
  #include <stdlib.h> /* defines NULL, getenv() */
  #include <sys/wait.h>      /* needed for wait() */
  #include <sys/time.h>    /* needed for gettimeofday(), struct timeval */
+                          /* and for setitimer(), struct itimerval */
+ #include <time.h>        /* nanosleep(), struct timespec */
+ #include <signal.h>      /* signal(), kill(), USR1, USR2, STOP, CONT */
+ #include <stdbool.h>      /* bool, true, false */

+ #define UNUSED __attribute__((unused))
  #define MAXBUF 4096 /* maximum size of command + args */
  #define MAXPCBS 512 /* maximum number of child processes */
  #define MAXARGS 128 /* maximum number of arguments in each command */
+ #define MAXCORES 32     /* maximum number of cores handled */
+ #define MIN_QUANTUM 100
+ #define MAX_QUANTUM 1000
+ #define TICKS_IN_QUANTUM 5

  typedef struct pcb {
      pid_t pid;          /* the pid associated with that process */
+     int ticks;         /* ticks left before quantum expires */
+     bool isalive;      /* true when created, false when SIGCHLD received */
+     bool sendusr1;     /* USR1 should be sent to start the process */
  } PCB;
```

```
  /*
***************
```

This indicates that **thv3** has an additional **#include** directive for **"ADTs/arrayqueue.h"**. It also defines maximum number of cores, minimum and maximum quantum values, and the number of ticks in a quantum. Three additional fields are added to the PCB structure.

```
*** 20,26 ****
  int quantum = 0;
  int nprocesses = 0;
  int ncores = 0;
! int active_processes = 0;

  static void print_usage_and_exit(char *name) {
      p1putstr(2, "usage: ");
--- 33,133 ----
  int quantum = 0;
  int nprocesses = 0;
  int ncores = 0;
! volatile int active_processes = 0;
! volatile int USR1_seen = 0;
! pid_t mypid;
! const Queue *readyQ = NULL;
! PCB *current[MAXCORES];
! int sub_quantum = 0;
!
! /*
!  * SIGUSR1 handler
!  */
! static void onusr1(UNUSED int sig) {
!     USR1_seen++;
! }
!
! /*
!  * SIGUSR2 handler
!  */
! static void onusr2(UNUSED int sig) {
! }
!
! /*
!  * map pid to index into pcb_array
!  */
! static int pid2index(pid_t pid) {
!     int i;
!
!     for (i = 0; i < num_procs; i++)
!         if (pcb_array[i].pid == pid)
!             return i;
!     /* SHOULD NEVER GET HERE */
!     return -1;
! }
!
! /*
!  * SIGCHLD handler
!  */
! static void onchld(UNUSED int sig) {
!     pid_t pid;
!     int status;
!
!     /*
!      * wait for all dead processes.
!      * we use a non-blocking call to be sure this signal handler will not
```

```
!      * block if a child was cleaned up in another part of the program.
!      */
!     while ((pid = waitpid(-1, &status, WNOHANG)) > 0) {
!         if (WIFEXITED(status) || WIFSIGNALED(status)) {
!             pcb_array[pid2index(pid)].isalive = 0;
!             active_processes--;
!             kill(mypid, SIGUSR2);     /* wake up pause */
!         }
!     }
! }
!
! /*
!  * SIGALRM handler
!  */
! static void onalrm(UNUSED int sig) {
!     /* this is the scheduler */
!     static bool first_time = true;
!     int count;
!
!     if (! first_time) {
!         int i;
!
!         for (i = 0; i < ncores; i++) {
!             PCB *p = current[i];
!             if ( p != NULL && p->isalive) {
!                 if (--(p->ticks) > 0)
!                     continue;                /* quantum not yet expired */
!                 (void) kill(p->pid, SIGSTOP);
!                 readyQ->enqueue(readyQ, ADT_VALUE(p));
!             }
!             current[i] = NULL;
!         }
!     } else
!         first_time = false;
!     for (count = 0; count < ncores; count++) {
!         PCB *p;
!         if (current[count] != NULL)
!             continue;
!         while (readyQ->dequeue(readyQ, ADT_ADDRESS(&p))) {
!             if (! p->isalive)
!                 continue;
!             p->ticks = TICKS_IN_QUANTUM;
!             if (p->sendusr1) {
!                 p->sendusr1 = false;
!                 (void) kill(p->pid, SIGUSR1);
!             } else
!                 (void) kill(p->pid, SIGCONT);
!             current[count] = p;
!             break;
!         }
!     }
! }

  static void print_usage_and_exit(char *name) {
      p1putstr(2, "usage: ");
***************
```

This indicates that **thv3** has four additional global variables: **mypid** (the parent process's pid for use with **SIGUSR2**), **readyQ** (holds processes waiting for their turn at the CPUs), **current[]** (the currently executing child processes), and **sub_quantum** (I have divided the quantum into several sub quanta, this is the value of the sub quantum). Additionally, several static functions and signal handlers are defined:

- **onusr2()** – the **SIGUSR2** handler, does nothing, just returns
- **pid2index()** performs a linear search through **pcb_array** to find the PCB that corresponds to the **pid** argument – this is called by the **SIGCHLD** handler when it determines that a child process has terminated.
- **onchld()** – the **SIGCHLD** handler, lifted directly from JustCH8.pdf with one exception – it signals **SIGUSR2** to the parent process so the parent process can accurately keep track of the termination of its children.
- **onalrm()** – the **SIGALRM** handler, which is invoked every time the interval timer fires. It checks to see if this is the first time it is called; if not, it checks each currently running process to see if it is still alive; if so, it decrements the ticks of sub quanta; if this is still non-zero, it leaves the process running; otherwise it sends the **STOP** signal to the process and adds it to the **readyQ**.

  Then, for each processor, if there is no longer a running process on that processor, It removes the PCB at the head of the **readyQ**; if that process is no longer alive, it fetches another PCB; when it has a live one, it initializes the **ticks** field; if this is the first time we are starting this process (**sendusr1** is true), we set **sendusr1** to false and send **SIGUSR1** to the process; otherwise, we send **SIGCONT** to the process.

```
*** 66,71 ****
--- 173,180 ----
      PCB *p = pcb_array;
      char *args[MAXARGS];
      struct timeval start, stop;
+     struct timespec ms20 = {0, 20000000};      /* 20 ms */
+     struct itimerval it_val;

  /*
   * process environment variables and command argument to th?
***************
```

This indicates that **thv3** has one additional automatic variable in main(), a **struct itimerval**, needed when we start the interval timer.

```
*** 108,113 ****
--- 217,291 ----
          print_usage_and_exit(progname);
      }
  /*
+  * establish USR1 signal handler
+  */
+     if (signal(SIGUSR1, onusr1) == SIG_ERR) {
+         p1putstr(2, "Can't establish SIGUSR1 handler\n");
+         _exit(1);
+     }
+ /*
+  * establish USR2 signal handler
+  */
+     if (signal(SIGUSR2, onusr2) == SIG_ERR) {
+         p1putstr(2, "Can't establish SIGUSR2 handler\n");
+         _exit(1);
+     }
+     mypid = getpid();           /* used to send ourselves USR2 signal */
+ /*
+  * establish CHLD signal handler
+  */
+     if (signal(SIGCHLD, onchld) == SIG_ERR) {
```

```
+         p1putstr(2, "Can't establish SIGCHLD handler\n");
+         _exit(1);
+     }
+ /*
+  * establish ALRM signal handler
+  */
+     if (signal(SIGALRM, onalrm) == SIG_ERR) {
+         p1putstr(2, "Can't establish SIGALRM handler\n");
+         _exit(1);
+     }
+ /*
+  * make sure that the number of cores and quantum value are reasonable
+  */
+     if (ncores > MAXCORES) {
+         p1putstr(2, "Number of cores specified(");
+         p1putint(2, ncores);
+         p1putstr(2, ") is greater than the maximum value(");
+         p1putint(2, MAXCORES);
+         p1putstr(2, "), setting to maximum value\n");
+         ncores = MAXCORES;
+     }
+     for (n = 0; n < ncores; n++)
+         current[n] = NULL;
+     if (quantum < MIN_QUANTUM) {
+         p1putstr(2, "Quantum specified(");
+         p1putint(2, quantum);
+         p1putstr(2, ") is less than minimum value (");
+         p1putint(2, MIN_QUANTUM);
+         p1putstr(2, "), setting to minimum value\n");
+         quantum = MIN_QUANTUM;
+     } else if (quantum > MAX_QUANTUM) {
+         p1putstr(2, "Quantum specified(");
+         p1putint(2, quantum);
+         p1putstr(2, ") is greater than maximum value (");
+         p1putint(2, MAX_QUANTUM);
+         p1putstr(2, "), setting to maximum value\n");
+         quantum = MAX_QUANTUM;
+     }
+ /*
+  * make quantum be a multiple of 100 ms and set up sub-quantum
+  */
+     quantum = 100 * ((quantum + 1) / 100);
+     sub_quantum = quantum / TICKS_IN_QUANTUM;
+ /*
+  * create the ready queue
+  */
+     if ((readyQ = ArrayQueue(0L, doNothing)) == NULL) {
+         p1putstr(2, "Error creating the ready queue\n");
+         _exit(1);
+     }
+ /*
   * at this point, we know:
   *    the command to execute in each process
   *    the number of processes to create running the command
***************
```

This indicates that **thv3** has additional code to establish the **SIGUSR2** handler, fill in the value of **mypid**, establish the **SIGCHLD** handler, establish the **SIGALRM** handler, perform a sanity check on the quantum value and the number of cores supplied by the user, compute the subquantum value, and creates the **readyQ**.

```
*** 132,137 ****
```

```
--- 309,316 ----
                       _exit(1);
                       break;
              case 0: /* child branch */
+                     while (! USR1_seen)
+                         (void) nanosleep(&ms20, NULL);
                      execvp(args[0], args);
                      p1putstr(2, "Child process: error execvp'ing ");
                      p1putstr(2, cmd);
***************
```

We saw this for **thv2** for the child process to wait for the **USR1** signal.

```
*** 132,161 ****
                       _exit(1);
                       break;
              case 0: /* child branch */
                      execvp(args[0], args);
                      p1putstr(2, "Child process: error execvp'ing ");
                      p1putstr(2, cmd);
                      p1putstr(2, "\n");
                      for (j = 0; args[j] != NULL; j++)
                          free(args[j]);
                      _exit(1);
                      break;
              default:/* parent branch */
                      p->pid = pid;
                      p++;
          }
          num_procs++;
      }
      active_processes = num_procs;
    /*
     * now wait for all child processes to terminate
     */
      while (active_processes > 0) {
!         (void) wait(NULL);/* wait for a child process to terminate */
!         active_processes--;
      }
      (void)gettimeofday(&stop, NULL);
      print_timing(&start, &stop, cmd, nprocesses, ncores);
      for (j = 0; args[j] != NULL; j++)
          free(args[j]);
    /*
--- 310,364 ----
                       _exit(1);
                       break;
              case 0: /* child branch */
+                     while (! USR1_seen)
+                         (void) nanosleep(&ms20, NULL);
                      execvp(args[0], args);
                      p1putstr(2, "Child process: error execvp'ing ");
                      p1putstr(2, cmd);
                      p1putstr(2, "\n");
                      for (j = 0; args[j] != NULL; j++)
                          free(args[j]);
+                     readyQ->destroy(readyQ);
                      _exit(1);
                      break;
              default:/* parent branch */
                      p->pid = pid;
+                     p->isalive = true;
+                     p->sendusr1 = true;
+                     (void) readyQ->enqueue(readyQ, p);
```

```
                      p++;
+                     break;
           }
           num_procs++;
       }
       active_processes = num_procs;
   /*
+   * now start interval timer
+   */
+      it_val.it_value.tv_sec = sub_quantum/1000;
+      it_val.it_value.tv_usec = (sub_quantum*1000) % 1000000;
+      it_val.it_interval = it_val.it_value;
+      if (setitimer(ITIMER_REAL, &it_val, NULL) == -1) {
+          p1perror(2, "error calling setitimer()");
+          for (i = 0; i < num_procs; i++)
+              (void) kill(pcb_array[i].pid, SIGKILL);
+          goto cleanup;
+      }
+      onalrm(SIGALRM);            /* schedule the first set of processes */
+ /*
    * now wait for all child processes to terminate
    */
       while (active_processes > 0) {
!          pause();                /* wait for SIGUSR2 to wake us up */
       }
       (void)gettimeofday(&stop, NULL);
       print_timing(&start, &stop, cmd, nprocesses, ncores);
+ /*
+   * now clean up after ourselves, destroying the queue and returning
+   * all heap-allocated storage
+   */
+ cleanup:
+      readyQ->destroy(readyQ);
       for (j = 0; args[j] != NULL; j++)
           free(args[j]);
   /*
***************
```

Several changes here: in the parent branch of the fork() call, we initialize **isalive** to true, **sendusr1** to true, and add the PCB to the **readyQ**. After all processes have been created, we start the interval timer; if we fail to create the interval timer, we have to kill all of the child processes. We start the first processes by simply calling the **SIGALRM** handler function from main. Since the **SIGCHLD** handler is harvesting children as they terminate, all the mainline has to do to wait for all children to terminate is replace our previous call to **wait()** with a call to **pause()**, which pauses the thread until the process receives a signal; in particular, we have structured the program so that a **USR2** signal will definitely wake it up. Finally, in the cleanup section, we need to destroy the **readyQ**.

thv4

Building on **thv3**, this version had to periodically access the **/proc** file system, extracting meaningful statistics for the child processes, and format this data and print them out. The handout was not prescriptive, other than that the output should include something about the command being executed, execution time, memory used, and I/O.

The easiest way to meet this requirement is to collect the statistics for each process after it has been pre-empted, and print that line before scheduling the next set of processes. That is the approach taken in the code below.

My solution focuses on three files in **/proc/<pid>/**:

- **io** – obtain # of read system calls and write system calls (lines 3 and 4)
- **stat** – obtain state, page faults, user time, system time, virtual memory size, resident set size; stat consists of a single line of text, with each item separated from the others by a space; the desired items are in the 3rd, 12th, 14th, 15th, 23rd, and 24th fields, respectively (starting at 1)
- **cmdline** – the command and its args

Note that the virtual memory size is in bytes, while resident set size is in pages; I use **sysconf(_SC_PAGESIZE)** to convert the virtual memory size to pages. The user and system times are in clock ticks; I use **sysconf(_SC_CLK_TCK)** to convert these times to seconds. When outputting read system calls and write system calls, if the number gets too large, I divide it by 1000 and print the resulting number followed by **k** (for thousands).

I print these items out right justified (except for command line) in the order pid, state, virtual memory size, resident set size, page faults, system calls read, system calls write, user time, system time, command line in fields of width 6, 3, 5, 5, 5, 10, 10, 6, 6, 0 characters, respectively. At the beginning, and after every 20 lines of process output, I display the header line.

The changes needed to produce **thv4** from **thv3** are shown as the output produced by executing the following command:

% diff --context=3 uspsv3.c uspsv4.c

I provide commentary within the diff output describing what it means.

```
*** thv3.c     2022-10-03 10:35:16.477240599 -0700
--- thv4.c     2022-10-03 10:47:04.174705603 -0700
***************
*** 8,13 ****
--- 8,16 ----
  #include <time.h>       /* nanosleep(), struct timespec */
  #include <signal.h>     /* signal(), kill(), USR1, USR2, STOP, CONT */
  #include <stdbool.h>      /* bool, true, false */
+ #include <sys/types.h>  /* three includes needed for open() and O_RDONLY */
+ #include <sys/stat.h>
+ #include <fcntl.h>

  #define UNUSED __attribute__((unused))
  #define MAXBUF 4096 /* maximum size of command + args */
***************
```

In order to open the files in **/proc**, we need to **#include** three files: **<sys/types.h>**, **<sys/stat.h>**, and **<fcntl.h>**.

```
*** 87,92 ****
--- 90,259 ----
      }
  }

+ static long p1atol(char *s) {
+     long ans;
+
+     for (ans = 0l; *s >= '0' && *s <= '9'; s++)
+         ans = 10 * ans + (long)(*s - '0');
+     return ans;
+ }
+
+ static void p1ltoa(long number, char *buf) {
+     char tmp[25];
+     long n, i, negative;
+     static char digits[] = "0123456789";
+
+     if (number == 0l) {
+         tmp[0] = '0';
+         i = 1;
+     } else {
+         if ((n = number) < 0l) {
+             negative = 1;
+             n = -n;
+         } else
+             negative = 0;
+         for (i = 0; n != 0l; i++) {
+             tmp[i] = digits[n % 10l];
+             n /= 10l;
+         }
+         if (negative) {
+             tmp[i] = '-';
+             i++;
+         }
+     }
+     while (--i >= 0)
+         *buf++ = tmp[i];
+     *buf = '\0';
+ }
+
+ static void scopy(char *from, char *to) {
+     while ((*to++ = *from++) != '\0')
+         ;
+ }
+
+ /*
+  * display information about process on specified file descriptor
+  */
+ #define PRINT_LIMIT 20          /* print out header every 20 lines */
+ static void display(pid_t pid, int outfd) {
+     char filename[30], ibuf[4096], pidstr[10], word[50];
+     char obuf[200];
+     char *sp, *rsp;
+     int fd;
+     int n, i;
+     long pagesize = sysconf(_SC_PAGESIZE);
+     long ticks_per_sec = sysconf(_SC_CLK_TCK);
+     long utime, stime, vsize;
```

```
+     char spid[20], sstate[2], svsize[20], srsize[20], spgflts[20];
+     char ssyscr[20], ssyscw[20], sutime[20], sstime[20];
+     static char header[100];
+     static int init = 1;
+     static int nprinted = PRINT_LIMIT;
+
+     sp = filename;
+     sp = p1strpack("/proc/", 0, ' ', sp);
+     p1itoa((int)pid, pidstr);
+     sp = p1strpack(pidstr, 0, ' ', sp);
+     rsp = sp;
+     sp = p1strpack("/io", 0, ' ', sp);
+     if ((fd = open(filename, O_RDONLY)) == -1)
+         return;
+     (void)p1getline(fd, ibuf, sizeof ibuf);  /* skip bytes read */
+     (void)p1getline(fd, ibuf, sizeof ibuf);  /* skip bytes written */
+     if ((n = p1getline(fd, ibuf, sizeof ibuf)) == 0)
+         return;
+     ibuf[n-1] = '\0';
+     i = 0;
+     i = p1getword(ibuf, i, word);
+     i = p1getword(ibuf, i, word);
+     scopy(word, ssyscr);
+     if ((n = p1getline(fd, ibuf, sizeof ibuf)) == 0)
+         return;
+     ibuf[n-1] = '\0';
+     i = 0;
+     i = p1getword(ibuf, i, word);
+     i = p1getword(ibuf, i, word);
+     scopy(word, ssyscw);
+     close(fd);
+     sp = p1strpack("/stat", 0, ' ', rsp);
+     if ((fd = open(filename, O_RDONLY)) == -1)
+         return;
+     if ((n = p1getline(fd, ibuf, sizeof ibuf)) == 0)
+         return;
+     ibuf[n-1] = '\0';
+     for (i = 0, n = 1; i != -1; n++) {
+         i = p1getword(ibuf, i, word);
+         switch(n) {
+             case 1: scopy(word, spid); break;
+             case 3: scopy(word, sstate); break;
+             case 12: scopy(word, spgflts); break;
+             case 14: utime = p1atol(word) / ticks_per_sec;
+                      p1ltoa(utime, sutime); break;
+             case 15: stime = p1atol(word) / ticks_per_sec;
+                      p1ltoa(stime, sstime); break;
+             case 23: vsize = p1atol(word) / pagesize;
+                      p1ltoa(vsize, svsize); break;
+             case 24: scopy(word, srsize); break;
+             default: break;
+         }
+     }
+     close(fd);
+     sp = p1strpack("/cmdline", 0, ' ', rsp);
+     if ((fd = open(filename, O_RDONLY)) == -1)
+         return;
+     if ((n = p1getline(fd, ibuf, sizeof ibuf)) == 0)
+         return;
+     close(fd);
+     for (i = 0; i < n; i++) {
+         if (ibuf[i] == '\0') {
+             if (ibuf[i+1] != '\0')
```

```
+                 ibuf[i] = ' ';
+             else
+                 ibuf[i] = '\n';
+         }
+     }
+     if (init) {
+         init = 0;
+         sp = header;
+         sp = p1strpack("  PID ", -6, ' ', sp);
+         sp = p1strpack(" St", -3, ' ', sp);
+         sp = p1strpack(" VMSZ", -5, ' ', sp);
+         sp = p1strpack(" RSSZ", -5, ' ', sp);
+         sp = p1strpack(" FLTS", -5, ' ', sp);
+         sp = p1strpack("  SysCR", -10, ' ', sp);
+         sp = p1strpack("  SsyCW", -10, ' ', sp);
+         sp = p1strpack(" usrtm", -6, ' ', sp);
+         sp = p1strpack(" systm", -6, ' ', sp);
+         sp = p1strpack(" Cmd\n", 0, ' ', sp);
+     }
+     if (nprinted >= PRINT_LIMIT) {
+         p1putstr(outfd, header);
+         nprinted = 0;
+     }
+     if ((n = p1strlen(ssyscr)) > 6) {
+         ssyscr[n-3] = 'k';
+         ssyscr[n-2] = '\0';
+     }
+     if ((n = p1strlen(ssyscw)) > 6) {
+         ssyscw[n-3] = 'k';
+         ssyscw[n-2] = '\0';
+     }
+     sp = obuf;
+     sp = p1strpack(pidstr, -6, ' ', sp);
+     sp = p1strpack(sstate, -3, ' ', sp);
+     sp = p1strpack(svsize, -5, ' ', sp);
+     sp = p1strpack(srsize, -5, ' ', sp);
+     sp = p1strpack(spgflts, -5, ' ', sp);
+     sp = p1strpack(ssyscr, -10, ' ', sp);
+     sp = p1strpack(ssyscw, -10, ' ', sp);
+     sp = p1strpack(sutime, -6, ' ', sp);
+     sp = p1strpack(sstime, -6, ' ', sp);
+     sp = p1strpack(" ", 0, ' ', sp);
+     sp = p1strpack(ibuf, 0, ' ', sp);
+     p1putstr(outfd, obuf);
+     nprinted++;
+ }
+
  /*
   * SIGALRM handler
   */
```

p1atoi() in **p1fxns.h** only returns an integer, while some of the numbers involved in the display are longs. Thus **p1atol()** converts a string to a long integer.

p1itoa() in **p1fxns.h** only converts an integer, while some of the numbers involved in the display are longs. Thus **p1ltoa()** converts a long integer to a string.

Since **strcpy(3)** cannot be used in this project, **scopy()** is a simple function to copy one string to another.

display() displays information about **pid** on **outfd**. Note the use of static variables in the function so that they retain their values across invocations of **display()**.

We use **p1strpack()** to generate the filename needed in an **open()** invocation. If a file open or a read fails at any time in this routine, we simply return, as this likely means that the process has actually terminated, with the result that its entries in **/proc** no longer exist. There is no loss of functionality doing this, as there is nothing to report about a process that has terminated.

The code accumulates the statistics for the process from **io**, **stat**, and **cmdline**, formats them up into a single buffer, and prints the line on **outfd**. If we have printed 20 lines since last printing the header line, we print the header line, as well.

```
*** 105,110 ****
--- 272,278 ----
                          continue;                 /* quantum not yet expired */
                      (void) kill(p->pid, SIGSTOP);
                      readyQ->enqueue(readyQ, ADT_VALUE(p));
+                     display(p->pid, 1);
                 }
                 current[i] = NULL;
            }
***************
```

After the **STOP** signal is sent to a process and it has been added to the end of the **readyQ**, **display()** is called to report its statistics on standard output.