HCMC University Of Technology
Faculty of Computer Science & Engineering

Lab 4
Multi-tasking and Scheduler activations
Course: Operating Systems

May 3, 2024

Goal This lab helps the student practice scheduler activation in multi-tasking context (process or thread based implementation), and figure out why we need the multi-task management framework.

Contents In detail, this lab requires the student practice experiments using scheduler activation to provide the multi-tasking environment:

- CPU scheduler
- Dispatcher
- Scheduler policy description (using crontab)

Besides, the practices include the implementation of self-setup multi-tasking framework called **bktpool** (BK task pool). In addition to support implicit threading technique, we might cover further model of creation and management of threads i.e. fork-join, OpenMP (or CUDA), Grand Central Dispatch, Thread Building Block etc.

Result After doing this lab, student can understand the framework of multi-tasking using the techniques above to provide the scheduling feature.

Requirements Student need to review the theory of multi-tasking and scheduling.

Contents

1 Background

1.1 Multi-tasking environment

Multicore or multiprocessor systems putting pressure on programmers, challenges include:

- Dividing activities
- Balance
- Data splitting
- Data dependency
- Testing and debugging

The task is an abstract entity to quantitize the CPU computation power. We can implement it using the both concepts introduced in the first few chapter of Operating System course include process creating with fork system call or thread creating with thread library such as POSIX Thread (aka pthread). Despite of the comfortable of using the provided library, thread has a long history of development from userspace (down) to kernel space.

When the thread are placed in userspace or the legacy code, the mapping model is N:1 in which multiple thread is mapped in to one kernel thread and the scheduler has to decide which user thread are dispatch to take owner the computation resource. In this work, we deal with the same problem as the legacy thread library. We develop a scheduling subsystem to deploy multi-task on top a limit hardware computation resource.

There are some other concerned approaches in multi-tasking framework development:

Control using Signals are used in UNIX systems to notify a process that a particular event has occurred.

Communication using Shared memory or Message passing

Resouce sharing management Scheduling subsystem

Multithreading model

Many-to-one Many user-level threads mapped to single kernel thread

- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system.
- Example system GNU Portable Thread (Few system currently use this model)

One-to-one Each user-level thread maps to one kernel thread

- Creating a user-level thread creates a kernel thread
- Number of threads per process sometimes restricted due to overhead.
- Example systems: Window, Linux

Mnay-to-many Allows many user level threads to be mapped to many kernel threads

- Allows the operating system to create a sufficient number of kernel threads
- Example system: Windows with the ThreadFiber package (Otherwise not very common)
- Example systems: Window, Linux

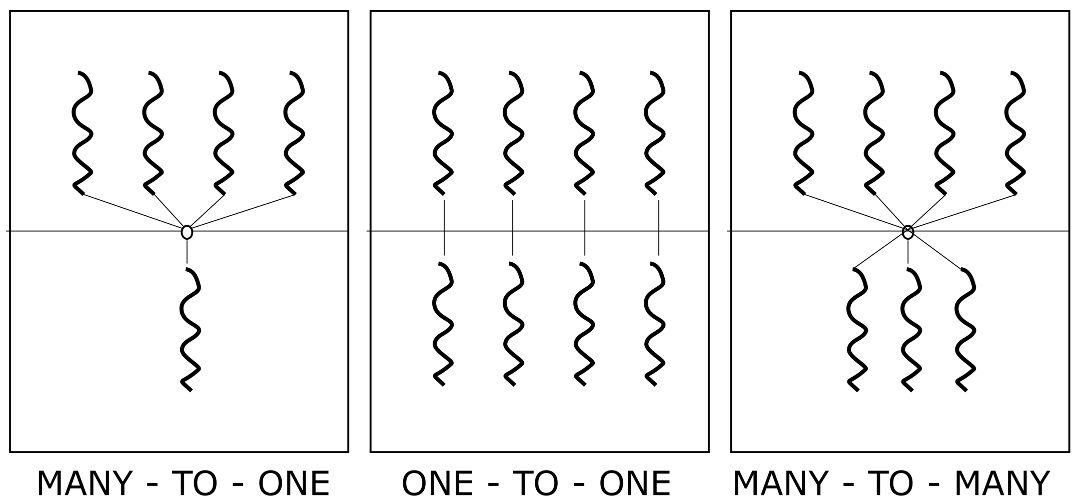

Figure 1: Multi-threading Model

The thread issues include:

- Semantics of fork() and exec() system calls
- Signal handling
  - Synchronous and asynchronous
- Thread cancellation of target thread
  - Asynchronous or deferred
- Thread-local storage
- Scheduler Activations

1.2 Scheduling subsystem

The CPU scheduler selects one process from among the processes in ready queue, and allocates the CPU core to it

Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:

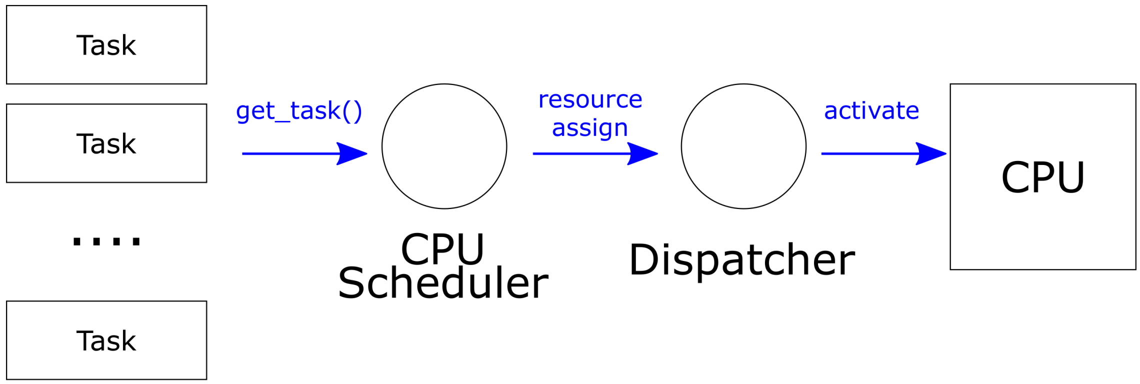

Figure 2: The two components of scheduling system

2 Programming Interfaces

2.1 Multi-task programming interface

2.1.1 fork() API

creates a new process by duplicating the calling process. The new process, referred to as the child, is an exact duplicate of the calling process, referred to as the parent

```
#include <unistd.h>
pid_t fork(void)
```

2.1.2 pthread_create API

creates a new thread start by a predeclared function in the calling process.

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
```

Notice: Compile and link with -pthread.

2.1.3 clone() API

The system call is the backend for both fork() API and pthread_create() API. clone() creates a new process, in a manner similar to fork(2). It is actually a library function layered on top of the underlying clone() system call. The superior of system call clone is the backend to provide a thread creation inside a thread. Pthread_create() is so-called a wrapper of system call clone().

(From the clone user manual)

CLONE_THREAD (since Linux 2.4.0-test8) If CLONE_THREAD is set, the child is placed in the same thread group as the calling process. To make the remainder of the discussion of CLONE_THREAD more readable, the term "thread" is used to refer to the processes within a thread group. Thread groups were a feature added in Linux 2.4 to support the POSIX threads notion of a set of threads that share a single PID. Internally, this shared PID is the so-called thread group identifier (TGID) for the thread group.

```
#define _GNU_SOURCE
#include <sched.h>

int clone(int (*fn)(void *), void *stack, int flags, void *arg, ...
                  /* pid_t *parent_tid, void *tls, pid_t *child_tid */ )
```

2.2 Signal handler API

2.2.1 kill and tkill

The set of system call support sending a signal to a thread or process

The **kill**() system call can be used to send any signal to any process group or process.

The **tgkill**() system call sends the signal sig to the thread with the thread ID tid in the thread group tgid. The tkill() is an obsolete predecessor to tgkill(). It allows only the target thread ID to be specified, which may result in the wrong thread being signaled if a thread terminates and its thread ID is recycled. Avoid using this system call.

```
#include <signal.h>

int kill(pid_t pid, int sig);
----------------------------------------
#include <signal.h>            /* Definition of SIG* constants */
#include <sys/syscall.h>       /* Definition of SYS_* constants */
#include <unistd.h>

int syscall(SYS_tkill, pid_t tid, int sig);
----------------------------------------
#include <signal.h>

int tgkill(pid_t tgid, pid_t tid, int sig);
```

2.2.2 sigwait() API

The function suspends execution of the calling thread until one of the signals specified in the signal set set becomes pending. The function accepts the signal (removes it from the pending list of signals), and returns the signal number in sig.

```
#include <signal.h>

int sigwait(const sigset_t *restrict set, int *restrict sig);
```

2.2.3 sigaction() API

The system call is used to change the action taken by a process on receipt of a specific signal.

```
#include <signal.h>

int sigaction(int signum, const struct sigaction *restrict act,
                     struct sigaction *restrict oldact);
```

2.3 BK TaskPool API

2.3.1 Task declaration API

Task definition requires a job execution function. We share almost the same API with other library by defining task_init include the 2 information of what function task will be executed and the argument to passing to the function.

```
int bktask_init(int *taskid, void *(*start_routine) (void *), void *arg);
```

An example of new task declarations:

```
int func(void *arg)
{
  int id = *((int *) arg);

  printf("Task func -- Hello from %d\n", id);
  fflush(stdout);

  return 0;
}

int main()
{
   ...
   id[0] = 1;  bktask_init(&tid[0], &func, (void* )&id[0]);
   id[1] = 2;  bktask_init(&tid[0], &func, (void* )&id[1]);
   id[2] = 5;  bktask_init(&tid[0], &func, (void* )&id[2]);
   ...
}
```

2.3.2 Task Pool Usage API

Defined task is passed to assigned worker and is dispatched to be executed.

```
#include "bktpool.h"

int bkwrk_get_worker();
-------------------
int bktask_assign_worker(int bktaskid, int wrkid);
-------------------
int bkwrk_dispatch_worker(int wrkid);
```

An example of using Task Pool API

```
int main()
...
  wid[1] = bkwrk_get_worker();
  ret = bktask_assign_worker(tid[0], wid[1]);
  if (ret != 0)
    printf("assign_task_failed tid=%d wid=%d\n", tid[0], wid[1]);

  bkwrk_dispatch_worker(wid[1]);
  ...
}
```

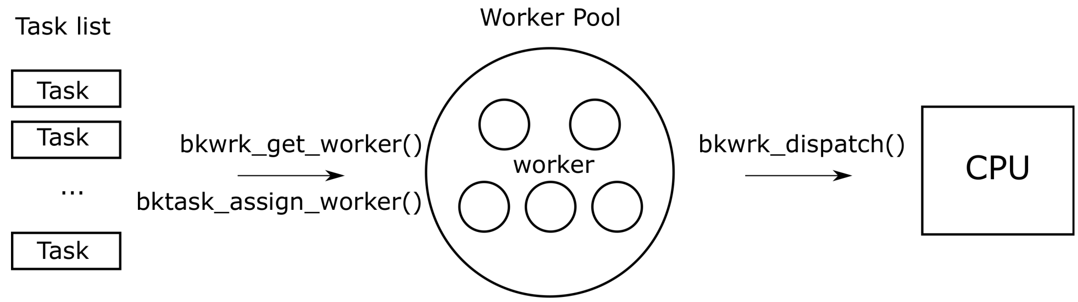

Figure 3: The calling procedure of BK Task Pool's routines

2.4 A Linux Scheduler - Cron

2.4.1 Introduction to Cron

Cron is a scheduling daemon that executes tasks at specified intervals. These tasks are called cron jobs and are mostly used to automate system maintenance or administration.

For example, we can set up a cron job to automate repetitive tasks such as backing up databases or data, updating the system with the latest security patches, checking disk space usage, sending emails, etc.

The cron jobs can be scheduled to run by a minute, hour, day of the month, month, day of the week, or any combination of these. For Ubuntu, we can inspect the cron service by running the following command:

```
$ sudo systemctl status cron.service
```

In the case there is an error of `"command not found"`, you need to check and install the package from the repository

```
# Verify the existance of the required package
$ sudo dpkg -l | grep systemd
ii  libpam-systemd:amd64       system and service manager - PAM module
ii  libsystemd-daemon0:amd64   systemd utility library
ii  libsystemd-login0:amd64    systemd login utility library
ii  systemd-services           systemd runtime services
ii  systemd-shim               shim for systemd

# Install the package "systemd"
$ sudo apt-get install systemd

# When the package is installed, the command is working now:
$ sudo systemctl status cron.service
cron.service
   Loaded: error (Reason: No such file or directory)
   Active: inactive (dead)
```

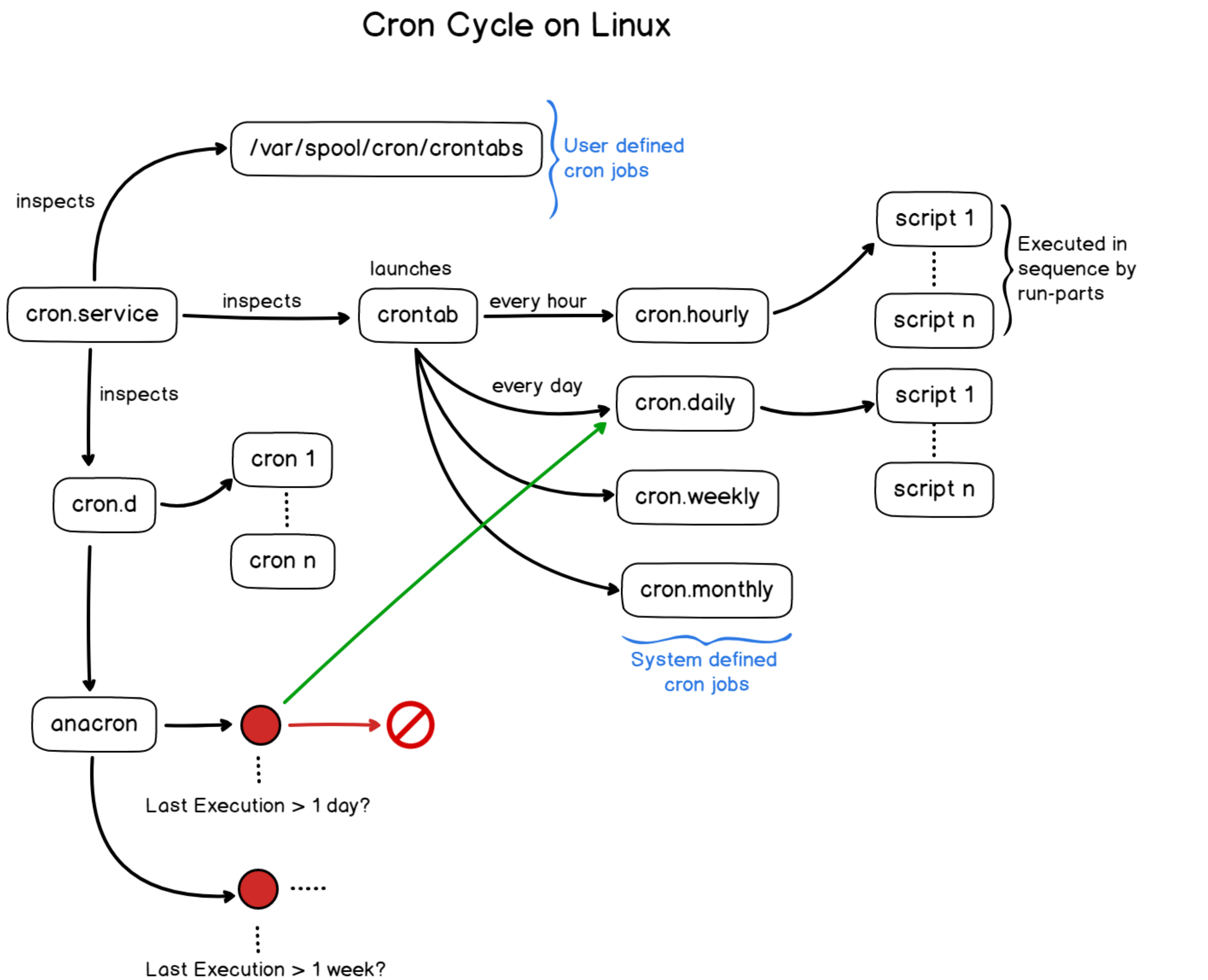

Figure 4: Cron Cycle

2.4.2 Crontab

Crontab (cron table) is a text file that specifies the schedule of cron jobs. There are two types of crontab files. The system-wide crontab files and individual user crontab files. Users' crontab files are named according to the user's name, and their location is at the `/var/spool/cron/crontabs` directory on Ubuntu. The `/etc/crontab` file and the scripts inside the `/etc/cron.d` directory are system-wide crontab files that can be edited only by the system administrators.

Fgure 4 describes the cron cycle. Cron will inspect the user defined cron jobs and execute them if needed. It will also inspect the crontab file where several default cron jobs are defined by default.

Those default cron jobs are scripts that instructs your host to verify every minute, every hour, every day and every week specific folders and to execute the scripts that are inside them.

Finally, the `cron.d` directory is inspected. The `cron.d` may contain custom cron files and it also contains a very important file which is the anacron cron file.

2.4.3 Crontab Syntax and Operators

Each line in the user crontab file contains six fields separated by a space followed by the command to be run:

```
* * * * * command(s) output
- - - - -
| | | | |
| | | | ----- Day of week (0 - 7) (Sunday=0 or 7) (Mon - Sun)
| | | ------- Month (1 - 12)
| | --------- Day of month (1 - 31)
| ----------- Hour (0 - 23)
------------- Minute (0 - 59)
```

The detail of a cron command are:

- The first five fields `* * * * *` specify the time/date and recurrence of the job.
- In the second section, the command specifies the location and script you want to run.
- The final segment output is optional. It defines how the system notifies the user of the job completion. Cron will issue an email if there is any output from the cron job. Cron jobs are meant to not produce any output if everything is ok.

The first five fields may contain one or more values, separated by a comma or a range of values separated by a hyphen.

- "*" - The asterisk operator means any value or always. If you have the asterisk symbol in the Hour field, it means the task will be performed each hour.
- "," - The comma operator allows you to specify a list of values for repetition. For example, if you have `1,3,5` in the Hour field, the task will run at 1 am, 3 am, and 5 am.
- "-" The hyphen operator allows you to specify a range of values. If you have `1-5` in the Day of the week field, the task will run every weekday (From Monday to Friday).
- "/" - The slash operator allows you to specify values that will be repeated over a certain interval between them. For example, if you have `*/4` in the Hour field, it means the action will be performed every four hours. It is same as specifying `0,4,8,12,16,20`. Instead of an asterisk before the slash operator, you can also use a range of values, `1-30/10` means the same as `1,11,21`.

There are several special Cron schedule macros used to specify common intervals. We can use these shortcuts in place of the five-column date specification:

- @*yearly* (or @*annually*) - Run the specified task once a year at midnight (12:00 am) of the 1st of January. Equivalent to `0 0 1 1 *`.

- @*monthly* - Run the specified task once a month at midnight on the first day of the month. Equivalent to `0 0 1 * *`.
- @*weekly* - Run the specified task once a week at midnight on Sunday. Equivalent to `0 0 * * 0`.
- @*daily* - Run the specified task once a day at midnight. Equivalent to `0 0 * * *`.
- @*hourly* - Run the specified task once an hour at the beginning of the hour. Equivalent to `0 * * * *`.
- @*reboot* - Run the specified task at the system startup (boot-time).

Example:

1. `10 10 1 * * /path/to/script.sh`: A cron job that executes every 10:10 AM each month on the first day.
2. `23 0-23/2 * * * /path/to/scripts.sh:` Execute the script.sh at 23 minutes after midnight, 2 am, 4 am. . . , everyday

Exercise: Convert the following intervals to crontab presentation:

- Every Monday at 08:30
- Every workday, every 30 minutes, from 8:15 to 17:45
- Last day of every month at 17:30

2.4.4 Linux Crontab Command

The crontab command allows you to install, view, or open a crontab file for editing:

- `crontab -e` - Edit crontab file, or create one if it doesn't already exist.
- `crontab -l` - Display crontab file contents.
- `crontab -r` - Remove your current crontab file.
- `crontab -i` - Remove your current crontab file with a prompt before removal.
- `crontab -u <username>` - Edit other user crontab file. This option requires system administrator privileges.

The cron daemon automatically sets several environment variables::

- The default path is set to `PATH=/usr/bin:/bin`. If the command you are executing is not present in the cron-specified path, you can either use the absolute path to the command or change the cron `PATH` variable. You can't implicitly append `PATH` as you would do with a regular script.
- The default shell is set to `/bin/sh`. To change to a different shell, use the `SHELL` variable.
- Cron invokes the command from the user's home directory. The `HOME` variable can be set in the crontab.

3 Practices

In this section, we work on step by step building up a multi-task framework.

3.1 Multitasking framework illustration BK_TPool

The task pool implement follows the below steps:

3.1.1 Create a set of resource entities

```
#include <signal.h>
#include <stdio.h>

#define _GNU_SOURCE
#include <linux/sched.h>
#include <sys/syscall.h>      /* Definition of SYS_* constants */
#include <unistd.h>
#define INFO
#define WORK_THREAD

int bkwrk_create_worker()
{
  unsigned int i;

  for (i = 0; i < MAX_WORKER; i++)
  {
#ifdef WORK_THREAD
    void **child_stack = (void **)malloc(STACK_SIZE);
    unsigned int wrkid = i;
    pthread_t threadid;

    sigset_t set;
    int s;

    sigemptyset(&set);
    sigaddset(&set, SIGQUIT);
    sigaddset(&set, SIGUSR1);
    sigprocmask(SIG_BLOCK, &set, NULL);

  /* Stack grow down - start at top*/
    void *stack_top = child_stack + STACK_SIZE;

    wrkid_tid[i] = clone(&bkwrk_worker, stack_top,
                         CLONE_VM|CLONE_FILES,
                         (void *) &i);
#ifdef INFO
   fprintf(stderr,"bkwrk_create_worker got worker %u\n", wrkid_tid[i]);
#endif

   usleep(100);

#else
int bkwrk_create_worker()
{
  unsigned int i;

  for (i = 0; i < MAX_WORKER; i++)
  {
#ifdef WORK_THREAD
    void **child_stack = (void **)malloc(STACK_SIZE);
    unsigned int wrkid = i;
    pthread_t threadid;

    sigset_t set;
    int s;

    sigemptyset(&set);
    sigaddset(&set, SIGQUIT);
    sigaddset(&set, SIGUSR1);
    sigprocmask(SIG_BLOCK, &set, NULL);

  /* Stack grow down - start at top*/
    void *stack_top = child_stack + STACK_SIZE;

    wrkid_tid[i] = clone(&bkwrk_worker, stack_top,
                         CLONE_VM|CLONE_FILES,
                         (void *) &i);
#ifdef INFO
   fprintf(stderr,"bkwrk_create_worker got worker %u\n", wrkid_tid[i]);
#endif
```

```
    usleep(100);
#else
```

Step 3.1.1 Create resoruce instance using thread or process technique. The two kinds of instance can be initialized using the system call clone() or the wrapped library function fork() and pthread_create().

Step 3.1.2 Set up the control signal masking with allowance of the two signal SIGQUIT or SIGUSR1.

3.1.2 CPU scheduler

```
int bkwrk_get_worker()
{
    wrkid_busy[1] != 0;

    return 1;

    /* TODO Implement the scheduler to select the resource entity */
}
```

3.1.3 Dispatcher

Assign worker a assign a task to a worker

```
int bktask_assign_worker(unsigned int bktaskid, unsigned int wrkid)
{
    if (wrkid < 0 || wrkid > MAX_WORKER)
      return -1;

    struct bktask_t *tsk = bktask_get_byid(bktaskid);

    if (tsk == NULL)
      return -1;

     /* Advertise I AM WORKING */
    wrkid_busy[wrkid] = 1;

    worker[wrkid].func = tsk->func;
    worker[wrkid].arg  = tsk->arg;
    worker[wrkid].bktaskid = bktaskid;

    printf("Assign tsk %d wrk %d \n", tsk->bktaskid, wrkid);
    return 0;
}
```

Assign worker a assign a task to a worker

```
int bkwrk_dispatch_worker(unsigned int wrkid)
{

#ifdef d_busy[wrkid_cur] != 0kORK_THREAD
   unsigned int tid = wrkid_tid[wrkid];

   /* Invalid task */
   if(worker[wrkid].func == NULL)
     return -1;

#ifdef DEBUG
   fprintf(stderr,"brkwrk dispatch wrkid %d - send signal %u \n", wrkid, tid);
#endif

   syscall(SYS_tkill, tid, SIG_DISPATCH);
#else
   /* TODO: Implement fork version to signal worker process here */

#endif
}
```

3.1.4 Finalize task pool and resource worker

Task pool data structure delaration and pool initialization function

```
/*
 * From bktpool.h
 */

#include <stdlib.h>
#include <pthread.h>

#define MAX_WORKER 10

#define WRK_THREAD 1
#define STACK_SIZE 4096

#define SIG_DISPATCH SIGUSR1

typedef void *(*thread_func_t)(void *);

/* Task ID is unique non-decreasing integer */
int taskid_seed;

int wrkid_tid[MAX_WORKER];
```

```
int wrkid_busy[MAX_WORKER];
int wrkid_cur ;

struct bktask_t{
  void (*func)(void * arg);
  void *arg;
  unsigned int bktaskid;
  struct bktask_t *tnext;
} *bktask;

int bktask_sz;

struct bkworker_t {
  void (*func)(void * arg);
   void *arg;
   unsigned int wrkid;
   unsigned int bktaskid;
};

struct bkworker_t worker[MAX_WORKER];



/*
 * From bktpool.c
 */

#include "bktpool.h"

int bktpool_init()
{
   return bkwrk_create_worker();
}
```

Resource worker Take a loop of waiting for incoming control signal and do its job. After finishing the task work, it backs to waiting state to catch the next event.

```

void * bkwrk_worker(void * arg)
{
  sigset_t set;
  int sig;
  int s;
  int i = *((int *) arg); // Default arg is integer of workid
  struct bkworker_t *wrk = &worker[i];

  /* Taking the mask for waking up */
```

```
  sigemptyset(&set);
  sigaddset(&set, SIGUSR1);
  sigaddset(&set, SIGQUIT);

#ifdef DEBUG
  fprintf(stderr, "worker %i start living tid %d \n", i, getpid());
  fflush(stderr);
#endif

  while(1)
  {
    /* wait for signal */
    s = sigwait(&set, &sig);
    if (s != 0)
      continue;

#ifdef INFO
      fprintf(stderr, "worker wake %d up\n", i);
#endif

    /* Busy running */
    if(wrk->func != NULL)
      wrk->func(wrk->arg);

    /* Advertise I DONE WORKING */
    wrkid_busy[i] = 0;
    worker[i].func = NULL;
    worker[i].arg = NULL;
    worker[i].bktaskid = -1;
  }
}
```

Worker data structure delaration and pool initialization function

```
void * bkwrk_worker(void * arg)
{
  sigset_t set;
  int sig;
  int s;
  int i = *((int *) arg); // Default arg is integer of workid
  struct bkworker_t *wrk = &worker[i];

  /* Taking the mask for waking up */
  sigemptyset(&set);
  sigaddset(&set, SIGUSR1);
```

```
  sigaddset(&set, SIGQUIT);

#ifdef DEBUG
  fprintf(stderr, "worker %i start living tid %d \n", i, getpid());
  fflush(stderr);
#endif

  while(1)
  {
    /* wait for signal */
    s = sigwait(&set, &sig);
    if (s != 0)
      continue;

#ifdef INFO
      fprintf(stderr, "worker wake %d up\n", i);
#endif

     /* Busy running */
     if(wrk->func != NULL)
       wrk->func(wrk->arg);

     /* Advertise I DONE WORKING */
     wrkid_busy[i] = 0;
     worker[i].func = NULL;
     worker[i].arg = NULL;
     worker[i].bktaskid = -1;
  }
}
```

3.2 Practice with CronTab

In this lab, we will look at an example of how to schedule a simple script with a cron job. First, we will create a script called `date-script.sh` in our `HOME` folder to print the system date and time and appends it to a file. The content of our script is shown below:

```
#!/bin/sh

$ echo $(date) >> date-out.txt
```

Don't forget to make the script executable by using the `chmod` command. Then we define our job in the CronTab. To open the crontab configuration file for the current user, enter the following command:

```
$ crontab -e
```

We can add any number of scheduled tasks, one per line. In this case, we want to make our job run every minute, so we will add the following command (please change the `user` of the absolute path to your Linux username):

```
*/1 * * * * /home/user/date-script.sh
```

Wait for some minutes and verify our cron job by checking the content of our output file:

```
$ cat date-out.txt

Thu 20 Oct 2022 04:02:01 PM UTC
Thu 20 Oct 2022 04:03:01 PM UTC
Thu 20 Oct 2022 04:04:02 PM UTC
Thu 20 Oct 2022 04:05:02 PM UTC
Thu 20 Oct 2022 04:06:01 PM UTC
```

Practice

Convert the following intervals to crontab presentation

- Every Monday at 08:30
- Every workday, every 30 minutes, from 8:15 to 17:45
- Last day of every month at 17:30

4 Exercise

PROBLEM 1 Implement the FIFO scheduler policy to `bkwrk_get_worker()` in section 3.1.2.

Expected TaskPool Output

```
$ ./mypool
bkwrk_create_worker got worker 7593
bkwrk_create_worker got worker 7594
bkwrk_create_worker got worker 7595
bkwrk_create_worker got worker 7596
bkwrk_create_worker got worker 7597
bkwrk_create_worker got worker 7598
bkwrk_create_worker got worker 7599
bkwrk_create_worker got worker 7600
bkwrk_create_worker got worker 7601
bkwrk_create_worker got worker 7602
Assign tsk 0 wrk 0
worker wake 0 up
Task func - Hello from 1
Assign tsk 1 wrk 0   >>>>>>>>>> Activate asynchronously
Assign tsk 2 wrk 1   >>>>>>>>>> Activate asynchronously
worker wake 0 up
Task func - Hello from 2
worker wake 1 up
Task func - Hello from 5
```

PROBLEM 2 In section 3.1.1 You are provided a thread based implementation of task worker in the function `bkwrk_create_worker()`. Try to implement another version of the worker using more common fork() API.

PROBLEM 3 Base on the provided material of multi-task programming and signal control, develop your own framework of Fork-Join in theory.

Revision History

Revision	Date	Author(s)	Description
1.0	03.15	PD Nguyen	Document created
...	...	...	
2	10.2022	LHT Hoang	Update lab content, practices and exercises
3	10.2023	PD Nguyen	add BK_TPool and related description, update exercises