HCMC University Of Technology
Faculty of Computer Science & Engineering

BK-ThreadPool and Scheduler Activation
Course: Operating Systems

Phuong-Duy Nguyen

April 13, 2023

Goal This document provides the description of 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. Pthread, Fork Join, OpenMP (or CUDA), Grand Central Dispatch, Thread Building Block etc.

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

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

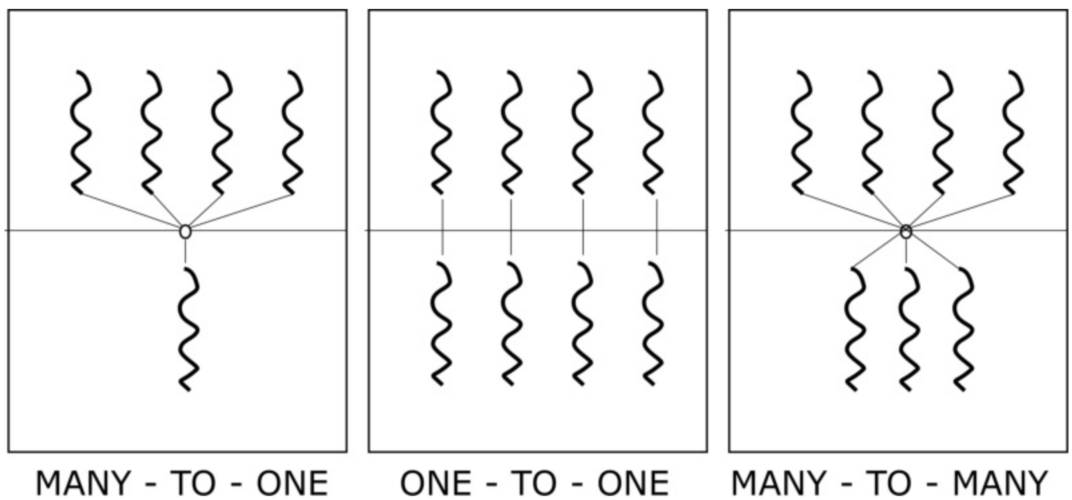

Figure 1: Multi-threading Model

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:

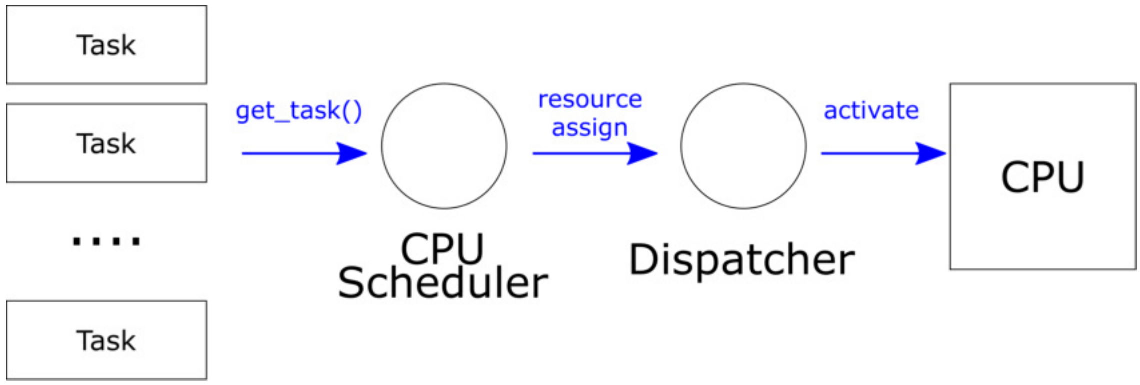

Figure 2: The two components of scheduling system

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 BK TaskPool API

2.2.1 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]);
   ...
}
```

2.2.2 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]);
  ...
}
```

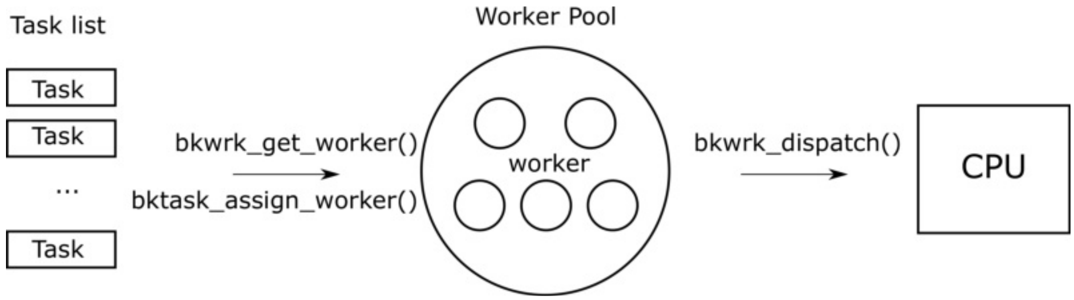

Figure 3: The calling procedure of BK Task Pool's routines

3 Implementations

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

```
/*
 * From bkwrk.c
 */
#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

      /* TODO: Implement fork version of create worker */
#endif

   }

   return 0;
}
```

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

```
/*
 * From bkwrk.c
 */
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 Assign a task to a worker

```
/*
 * From bkwrk.c
```

```
 */
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 Dispatch or activate the selected worker

```
/*
 * From bkwrk.c
 */
int bkwrk_dispatch_worker(unsigned int wrkid)
{

#ifdef WORK_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.

```
/*
 * From bkwrk.c
 */

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 The worker initialized function

```
/*
 * From bkwrk.c
 */

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;
  }
}
```

4 Exercises

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 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.

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

PROBLEM 4 Base on the provided material of multi-task programming and signal controllation, develop your own framework of Fork-Join in theory.

```
int fork(int taskid);
int join(int taskid);
```