Operating systems and concurrency - B04

David Kendall

Northumbria University

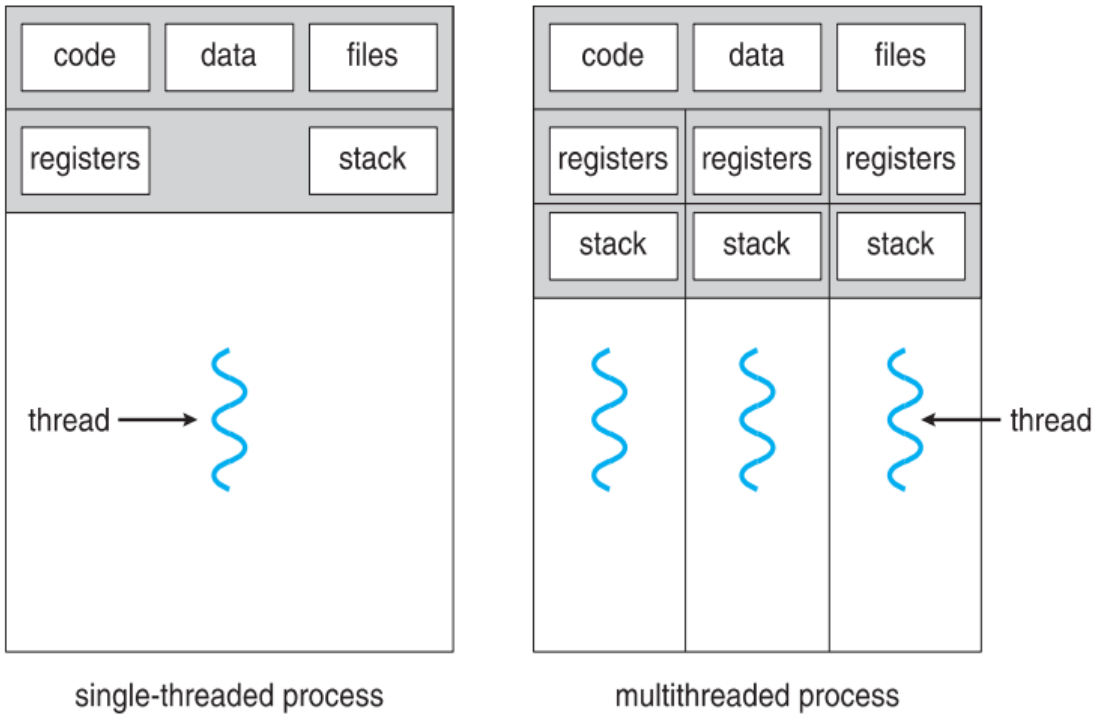
Introduction

- Introduction to threads
- Reminder of `fork()`
- `pthreads` example
- Comparison of `pthreads` with `fork()`
- Main `pthread` functions
- Some problems with threads

What are threads and why do we need them?

- We have already seen that it is very useful for the OS to provide real, or pseudo, concurrency
  - We can divide our work up into meaningful units that can be considered separately
  - When some unit of work is blocked waiting for I/O, another unit of work can make use of the CPU
- So far our unit of work is the *process*
- We can create multiple processes and allow the OS to schedule them to maximise the use of resources
- But the process is a *heavyweight* unit of work - it comes with lots of baggage: in addition to code, static data, registers, stack, heap etc. it also has open files, pipes, signals, sockets, devices etc
- The point of *threads* is to give the benefits of concurrency but in a much more *lightweight* form
- In fact, threads are sometimes called *lightweight processes*

Single- and multi-threaded processes



Source: SGG12, Chp. 4

- On the left is a ‘standard’ process with a single thread of control
- On the right is a multi-threaded process – a process that has created additional threads
- Notice what is shared and what is private

`fork()` reminder

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>

int globvar = 6;

int main(void) {
  int var = 88;
  pid_t pid;

  printf("before fork\n");
  if ((pid = fork()) < 0) {
      fprintf(stderr, "fork failed\n");
      exit(-1);
  } else if (pid == 0) {
      globvar++; //child
      var++;
      printf("Child : pid = %d, globvar = %d, var = %d\n",
               getpid(), globvar, var);
  } else {
      waitpid(pid, NULL, 0); // parent
      printf("Parent: pid = %d, globvar = %d, var = %d\n",
               getpid(), globvar, var);
  }
  exit(0);
}
```

pthreads example

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int globvar = 6;

void *threadController(void *arg) {
      globvar++;
      // var++; Notice this variable is not accessible
      printf("New thread : pid = %d, tid = 0x%lx, globvar = %d, "
              "var = %s\n", getpid(), (unsigned long)pthread_self(),
              globvar, "Not accessible");
      pthread_exit((void *)0);
}

int main(void) {
  int var = 88;
  pthread_t thread;

  printf("before thread create\n");
  if (pthread_create(&thread, NULL, threadController, NULL) != 0) {
      fprintf(stderr, "thread create failed\n");
      exit(-1);
  } else {
      pthread_join(thread, NULL); // main thread
      printf("Main thread: pid = %d, tid = 0x%lx, globvar = %d, "
             "var = %d\n", getpid(), (unsigned long)pthread_self(),
              globvar, var);
  }
  exit(0);
}
```

`fork()` compilation and output

```
$ gcc -o forkexample1 forkexample1.c
$ ./forkexample1
before fork
Child : pid = 9805, globvar = 7, var = 89
Parent: pid = 9804, globvar = 6, var = 88
```

`pthread` compilation and output

```
$ gcc -pthread -o threadexample1 threadexample1.c
$ ./threadexample1
before thread create
New thread : pid = 11030, tid = 0x7f12ddb66700, \
             globvar = 7, var = Not accessible
Main thread: pid = 11030, tid = 0x7f12de347740, \
             globvar = 7, var = 88
```

Points to notice in the example

- `threadexample` must be compiled with the option `-pthread`
  - this links the `pthread` library into the executable
- Process identifiers in `fork` example are *different*
- Process identifiers in `pthread` example are *the same*
- `globvar` and `var` in the parent and child processes are *separate*
- `globvar` in the main and new threads are *shared*
  - `var` is only accessible in the main thread, not in the new thread
- The main and new threads can be distinguished using the *thread identifiers* (`tid`)

Thread creation and destruction

- Create thread

```
#include <pthread.h>
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine) (void *),
                   void *arg);
Compile and link with -pthread.
```

  Creates `thread` with attributes `attr` and calls the `start_routine` with argument `arg`

- Terminate thread

```
void pthread_exit(void *retval);
```

  Can also just `return` from the thread. Terminates the calling thread and returns a value that is available to another thread in the same process that calls `pthread_join`

Thread join and self

- Thread join

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **retval);
```

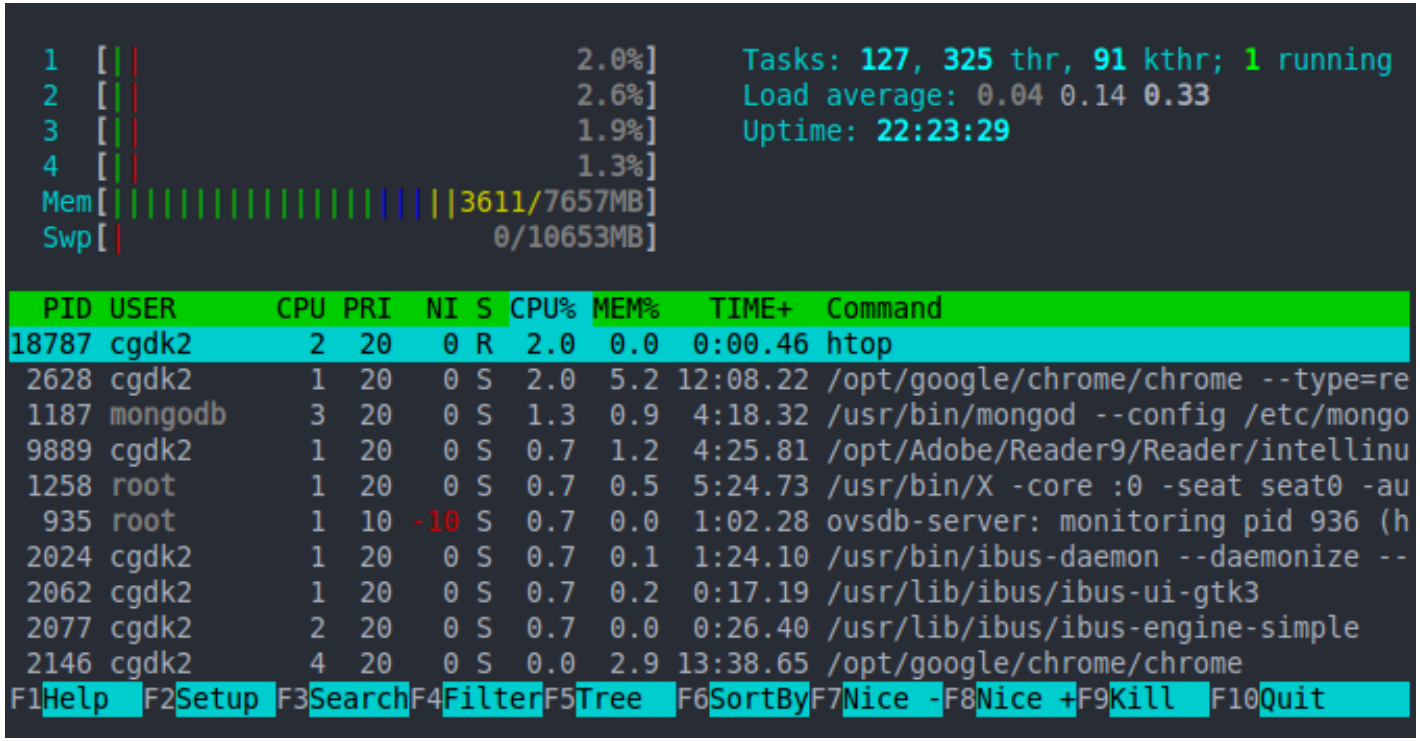
  Wait for `thread` to terminate. Return immediately if `thread` has already terminated. If `retval` is not `NULL` then copy then exit status of the target thread into the location pointed to by `*retval`

- Thread self

```
pthread_t pthread_self(void);
```

  Return the ID of the calling thread. This is the same value that is returned in `*thread` in the `pthread_create()` call that created this thread

Threads in action



- Output from `htop`
- Shows a machine with 4 cores
- Running 127 processes with 325 threads and 91 kernel threads
- 1 thread currently running

Some problems with threads

- Threads are not without their problems
- The main problems arise when multiple threads try to access shared data
- This can lead to *unpredictable* results
- The unpredictability arises because we don't know which thread will be chosen by the scheduler to execute next
- This leads to different instruction sequences
- and different instruction sequences can lead to different results!
- This has given rise to a whole discipline of concurrent programming with locks, semaphores, mutexes, condition variables etc. needed to recover predictable behaviour
- . . . more on this later in the module