CS2850 Operating System Lab

Week 9: Threads

nicolo colombo

nicolo.colombo@rhul.ac.uk

Office Bedford 2-21

Outline

Threads

POSIX threads

Mutexes

Threads

Computers are busy machines.

Threads help them perform several tasks at the same time.

A single process can solve *different related problems* by starting multiple threads.

Threads share the same *global* memory, e.g. data, but have their own stack (*automatic variables*).

Address space

A process memory contains

- process-wide resources, e.g. the program instructions and global data, and
- execution state information, e.g. the program counters and stack.

All threads can access process-wide resources but have private execution state information.

POSIX threads

There are differences between thread packages. Here we focus on Pthreads (for *POSIX* threads).

The Portable Operating System Interface (POSIX) is a set of *types and functions* that you can call from a C program (under UNIX).

Pthreads let you *divide* a program in *sub-tasks* and execute them sequentially or in parallel.

Pthreads in a C program

How to *create* multiple threads in C?

In a C program, you create new threads with a POSIX function called `pthread_create`.

You also need to

- define the task assigned to each thread (normally a subroutine of your program) and
- call a function that merges the output of threads when they return.

All threads are equal

There is *no hierarchy* between running threads, e.g. no parent-child distinctions.

Each thread executes independently: the actual execution order is *unpredictable*.

You synchronize them *explicitly* by waiting until they return.

Handling threads with POSIX *functions* requires to

- *include* the header `pthread.h` and
- *compile* your program with a `-pthread` flag.

Creating a Pthread

You create a new thread with[1]

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

- `thread` points to a *buffer* that stores the thread identifier,
- `attr` points to a `struct` that specifies various features of the new thread (write `NULL` for the default),
- `subroutine` points to the subroutine executed by the new thread, and
- `arg` points to the parameter of `start_routine`.

[1] See more about `pthread_create` on this page of the Linux online manual on Pthreads.

Terminating a Pthread

You wait for a thread to terminate and see its return value with

```
int pthread_join(pthread_t thread, void **retval)
```

- `thread` is the thread identifier and
- `retval` is the location where the exit status of the target thread is copied.

A thread terminates when it reaches a `return` statement or calls

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

Synchronization

You can ensure that one *event* (in one thread) happens before another event (in a concurrent thread) by synchronizing the two threads.

For example, you can wait for one thread to *return* by calling `pthread_join` in the main program.

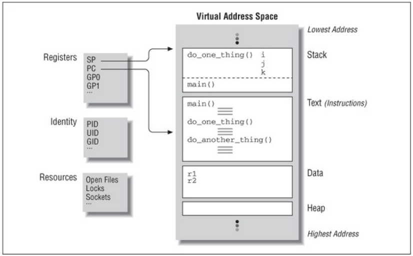
See more about thread synchronization in Chapter 3 of *PThreads Programming* by Farrel et al..

Threads or processes?

In the following examples, we implement the same program using

- no threads,
- a single thread,
- two threads, and
- two child processes.

A process with no threads



- Text contains the program instructions.
- Data are the global data needed to run the program.
- Heap is for dynamic memory allocation.
- Stack is for storing automatic variables.
- Registers, Identity, and Resources contain the info needed to execute.

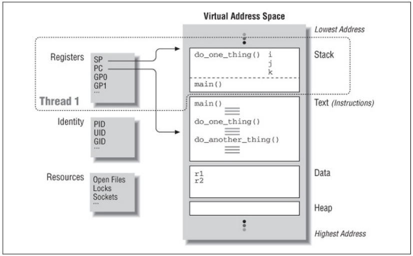
Example: main (simple version)

```
#include <stdio.h>
void sayGoodbye(int n1, int n2);
int brian(int n), dennis(int n);
int main(){
  int nBrian = 3, nDennis = 2;
  while(nBrian + nDennis > 0) {
    if (nBrian) nBrian = brian(nBrian);
    if (nDennis) nDennis = dennis(nDennis);
    sayGoodbye(nBrian, nDennis);
  }
}
int brian(int n) {
  for (int i = 0; i < n; i++) printf("%d-hi Dennis!\n", i + 1);
  return n - 1;
}
int dennis(int n) {
  for (int i = 0; i < n; i++) printf("%d-hi Brian!\n", i + 1);
  return n -1;
}
void sayGoodbye(int n1, int n2) {
    printf("%d-goodbye!\n", n1 + n2);
}
```

Output

```
 ./a.out
1-hi Dennis!
2-hi Dennis!
3-hi Dennis!
1-hi Brian!
2-hi Brian!
3-goodbye!
1-hi Dennis!
2-hi Dennis!
1-hi Brian!
1-goodbye!
1-hi Dennis!
0-goodbye!
```

A process with a single Pthreads



Stack and Registers are now part of the thread.

Each *active thread* has a reserved stack frame and specific machine registers.

Example: `main` (Pthread compatible)

`pthread_create` takes three free arguments

```
int pthread_create(pthread_t *tName, NULL, void *f(void
    *), void *arg)
```

`void *f(void *)` is (a pointer to) a function of type `void` which takes a pointer to `void` as a single argument.

The function arguments should be passed through a single pointer to `void`.

You need to rewrite `brian` and `dennis` as

```
void *brian(void *n)
void *dennis(void *n)
```

New version of `brian` and `dennis`

To meet the requirement of `pthread_create` you need a structure holding the original integer argument,

```
struct arg {
  int n;
};
```

In the first line of `f`, cast the `void` pointer to be a pointer to a variable of type `struct arg`[2]

```
  struct arg *p = n;
```

In main, `n` becomes `(&argName)->n`, where `argName` is an object of type `struct arg`.

[2] To access the original arguments with `p->n`.

New version of brian and dennis

You obtain

```
void *brian(void *nIn) {
  struct arg *nStack = nIn;
  for (int i=0; i<nStack->n; i++)
    printf("%d-hi Dennis!\n", i + 1);
  nStack->n = nStack->n - 1;
  return NULL;
}

void *dennis(void *nIn) {
  struct arg *nStack = nIn;
  for (int i=0; i<nStack->n; i++)
    printf("%d-hi Brian!\n", i + 1);
  nStack->n = nStack->n - 1;
  return NULL;
}
```

New version of `brian` and `dennis`

The new version of `main` is

```
#include <stdio.h>

struct arg{
  int n;
};

void *brian(void *n), *dennis(void *n), sayGoodbye(int n1, int n2);

int main(){
  struct arg nBrian = {3}, nDennis = {2};
  while(nBrian.n + nDennis.n > 0) {
    if (nBrian.n) brian(&nBrian);
    if (nDennis.n) dennis(&nDennis);
    sayGoodbye(nBrian.n, nDennis.n);
  }
}
```

`sayGoodbye` can be left unchanged.

Example (1 running Pthread)

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

struct arg{
  int n;
};

void *brian(void *n), *dennis(void *n), sayGoodbye(int n1, int n2);

int main(){
  pthread_t t1;
  struct arg nBrian = {3}, nDennis = {2};
  while(nBrian.n + nDennis.n > 0) {
    if (nBrian.n) pthread_create(&t1, NULL, brian, &nBrian);
    pthread_join(t1, NULL);
    if (nDennis.n) pthread_create(&t1, NULL, dennis, &nDennis);
    pthread_join(t1, NULL);
    sayGoodbye(nBrian.n, nDennis.n);
  }
}
```

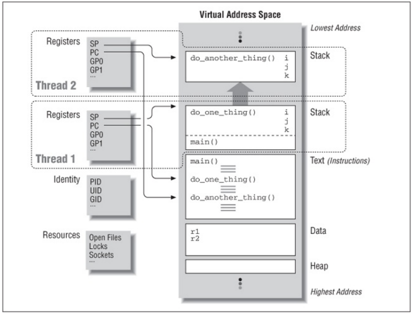
All auxiliary functions are defined as above.

Output (1 running Pthread)

The execution is as before

```
 ./a.out
1-hi Dennis!
2-hi Dennis!
3-hi Dennis!
1-hi Brian!
2-hi Brian!
3-goodbye!
1-hi Dennis!
2-hi Dennis!
1-hi Brian!
1-goodbye!
1-hi Dennis!
0-goodbye!
```

A process with two threads



Both threads have *their private copy* of the machine registers and can use variables or file descriptors in the *process-wide areas*.

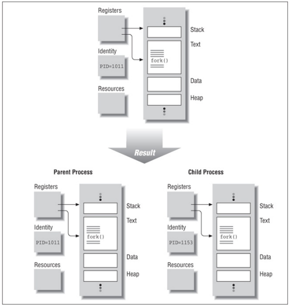
The two threads execute at different locations.

Example (2 running Pthreads)

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

struct arg{
        int n;
};

void *brian(void *n), *dennis(void *n), sayGoodbye(int n1, int n2);

int main(){
  struct arg nBrian = {3}, nDennis = {2};
  pthread_t t1, t2;
  while(nBrian.n + nDennis.n > 0) {
    if (nBrian.n) pthread_create(&t1, NULL, brian, &nBrian);
    if (nDennis.n) pthread_create(&t2, NULL, dennis, &nDennis);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    sayGoodbye(nBrian.n, nDennis.n);
  }
}
```

As `t1` starts before `t2` and the task performed by `brian` is *computationally cheap*, the output on `stdout` is unchanged.

Two processes vs. two threads



The address space of two concurrent processes created by `fork` is *completely private* (no shared areas).

The two processes can interact via a channel (anonymous pipe).

Example: main (2 running child processes, slowBrian)

```
int main() {
  int status;
  while(nBrian.n + nDennis.n > 0) {
    int pidBrian = fork();
    if (!pidBrian) {
      if (nBrian.n) brianSlow(&nBrian);
      return nBrian.n;
    }
    int pidDennis = fork();
    if (!pidDennis) {
      if (nDennis.n) dennis(&nDennis);
      return nDennis.n;
    }
    waitpid(pidBrian, &status, 0);
    nBrian.n = WEXITSTATUS(status);
    waitpid(pidDennis, &status, 0);
    nDennis.n = WEXITSTATUS(status);
    sayGoodbye(nBrian.n, nDennis.n);
  }
}
```

Include `unistd.h` and `sys/wait.h` to use `fork` and `wait`.

Output (2 running child processes, slow Brian)

The output is the same as before

```
 ./a.out
1-hi Dennis!
1-hi Brian!
2-hi Brian!
2-hi Dennis!
3-hi Dennis!
3-goodbye!
1-hi Dennis!
1-hi Brian!
2-hi Dennis!
1-goodbye!
1-hi Dennis!
0-goodbye!
```

Another example of synchronization: Mutexes

Variables of type `pthread_mutex_t` allow you to synchronize the access to some *area of your code*.

The protected area is mutually exclusive. While one thread is in the protected area, other threads cannot

- *execute* the protected code and
- *lock or unlock* the Mutex.

Mutex-protected variables

To ue a Mutex to regulate the *updates of shared variable*, `x`,

1. create and initialize a mutex variable, `m`, using
`pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;`
(`m` should be declared outside `main`).

2. in the thread subroutine, *lock and unlock* the mutex before updating `x`, e.g. write

```
pthread_mutex_lock(&m)
x= ...
pthread_mutex_unlock(&m)
```

Example: (2 running threads with thread-safe subroutines)

The *thread-safe* version of `slowBrian` and `dennis` is

```
void *brianSlow(void *n) {
  struct arg *n1 = n;
  int t = 0;
  pthread_mutex_lock(&m); //start of the mutual exclusive area
  for (int i=0; i<(n1->n); i++) {
    printf("%d-hi Dennis!\n", i + 1);
    while (t < 100000) t++;
  }
  pthread_mutex_unlock(&m);//end of the mutual exclusive area
  n1->n = n1->n - 1;
  return NULL;
}

void *dennis(void *n) {
  struct arg *n1 = n;
  pthread_mutex_lock(&m);//start of the mutual exclusive area
  for (int i=0; i<(n1->n); i++) printf("%d-hi Brian!\n", i + 1);
  pthread_mutex_unlock(&m);//end of the mutual exclusive area
  (n1->n) = (n1->n) -1;
  return NULL;
}
```

Output (2 running threads, thread-safe subroutines)

```
./a.out
1-hi Dennis!
2-hi Dennis!
3-hi Dennis!
1-hi Brian!
2-hi Brian!
3-goodbye!
1-hi Dennis!
2-hi Dennis!
1-hi Brian!
1-goodbye!
1-hi Dennis!
0-goodbye!
```