Processes - Part II

Amir H. Payberah
payberah@kth.se
2022

Threads

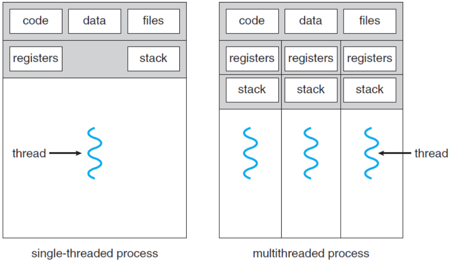
Thread

A basic unit of CPU utilization.

https://tinyurl.com/e8crhtne

Threads (1/2)

- A traditional process: has a single thread.
- Multiple threads in a process: performing more than one task at a time.
- Threads in a process share code section, data section, and other OS resources, e.g., open files.

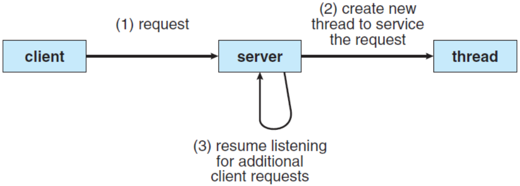


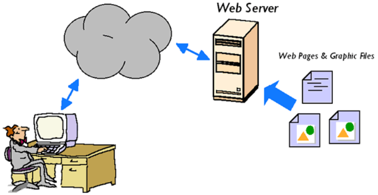
Threads (2/2)

- Multiple tasks of an application can be implemented by separate threads.
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request

Threads - Example

- Multi-threaded web-server architecture





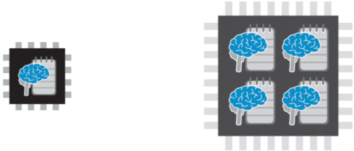
Threads Benefits

- Responsiveness: allow continued execution if part of process is blocked.
- Resource Sharing: threads share resources of process, easier than shared memory or message passing.
- Economy: thread switching has lower overhead than context switching.
- Scalability: process can take advantage of multiprocessor architectures.

Multi-core Programming

Multi-core Systems

- Users need more computing performance: single-CPU → multi-CPU
- A similar trend in system design: multi-core systems
  - Each core appears as a separate processor.

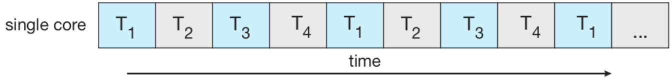


- Multi-threaded programming
  - Improves concurrency and more efficient use of multiple cores.

Concurrency vs. Parallelism (1/2)

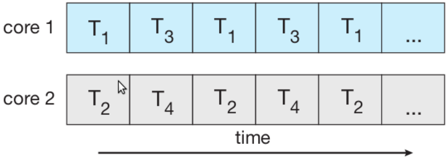
- Concurrency: supporting more than one task by allowing all the tasks to make progress.
  - A scheduler providing concurrency.

- Concurrent execution on a single-core system.



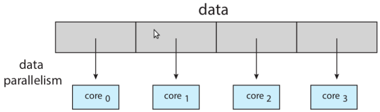
Concurrency vs. Parallelism (2/2)

- Parallelism: performing more than one task simultaneously.
- Parallelism on a multi-core system.

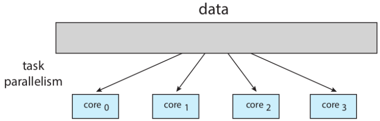


Types of Parallelism

- **Data** parallelism
  - Distributes subsets of the same data across multiple cores, same operation on each.



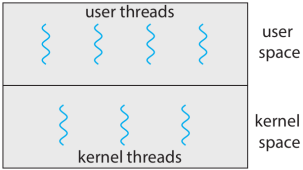
- **Task** parallelism
  - Distributes threads across cores, each thread performing unique operation.



Multi-threading Models

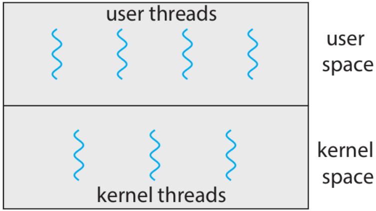
User Threads and Kernel Threads

- User threads: managed by user-level threads library.
  - Three primary thread libraries:
  - POSIX pthreads
  - Windows threads
  - Java threads
- Kernel threads: supported by the Kernel.



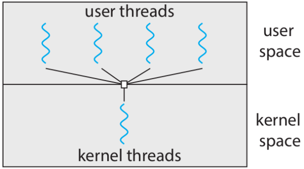
Multi-Threading Models

- Many-to-One
- One-to-One
- Many-to-Many



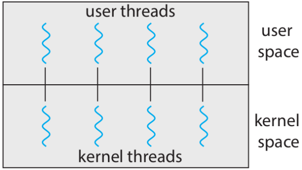
Many-to-One Model

- Many user-level threads mapped to single kernel thread.
- One thread blocking causes all to block.
- Multiple threads may not run in parallel on multi-core system because only one may be in kernel at a time.
- Few systems currently use this model.
  - Solaris green threads
  - GNU portable threads



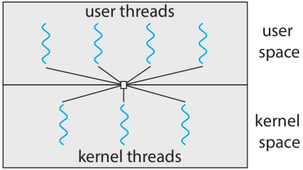
One-to-One Model

- Each user-level thread maps to one kernel thread.
- Creating a user-level thread creates a kernel thread.
- More concurrency than many-to-one.
- Number of threads per process sometimes restricted due to overhead.
- Examples:
  - Windows
  - Linux



Many-to-Many Model

- Allows many user-level threads to be mapped to many kernel threads.
- Allows the OS to create a sufficient number of kernel threads.
- Examples:
  - Windows with the ThreadFiber package
  - Otherwise not very common



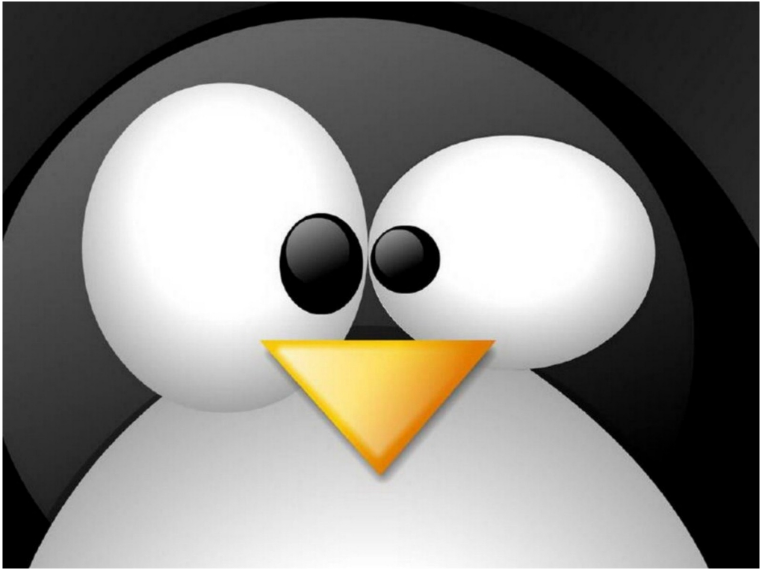
Thread Libraries

Thread Libraries (1/2)

- Thread library provides programmer with API for creating and managing threads.
- Two primary ways of implementing:
  - Library entirely in user-space.
  - Kernel-level library supported by the OS.

Thread Libraries (2/2)

- Pthread
  - Either a user-level or a kernel-level library.
- Windows thread
  - Kernel-level library.
- Java thread
  - Uses a thread library available on the host system.



Pthreads

- A POSIX API for thread creation and synchronization.
- Specification, not implementation.
- API specifies behavior of the thread library, implementation is up to development of the library.
- Common in UNIX OSs, e.g., Solaris, Linux, Mac OS X

Thread ID

- The thread ID (TID) is the thread analogue to the process ID (PID).
- The PID is assigned by the Linux kernel, and TID is assigned in the Pthread library.
- Represented by `pthread_t`.
- Obtaining a TID at runtime:

```
#include <pthread.h>

pthread_t pthread_self(void);
```

Creating Threads

- `pthread_create()` defines and launches a new thread.

```
#include <pthread.h>

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

- `thread_func` has the following signature:

```
void *thread_func(void *arg);
```

Terminating Threads

- Terminating yourself by calling `pthread_exit()`.

```
#include <pthread.h>

void pthread_exit(void *retval);
```

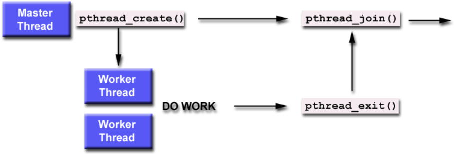
- Terminating others by calling `pthread_cancel()`.

```
#include <pthread.h>

int pthread_cancel(pthread_t thread);
```

Joining and Detaching Threads

- Joining allows one thread to block while waiting for the termination of another.
- You use join if you care about what value the thread returns when it is done, and use detach if you do not.

```
#include <pthread.h>

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



[https://computing.llnl.gov/tutorials/pthreads/#Joining]

A Threading Example

```c
void *thread_func(void *message) {
  printf("%s\n", (const char *)message);
  return message;
}

int main(void) {
  pthread_t thread1, thread2;
  const char *message1 = "Thread 1";
  const char *message2 = "Thread 2";

  // Create two threads, each with a different message.
  pthread_create(&thread1, NULL, thread_func, (void *)message1);
  pthread_create(&thread2, NULL, thread_func, (void *)message2);

  // Wait for the threads to exit.
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);

  return 0;
}
```

Implicit Threading

Implicit Threading

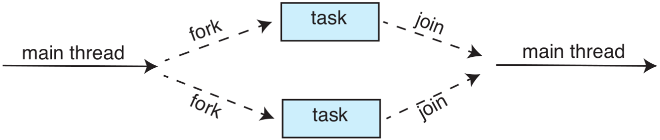
- Increasing the number of threads: program correctness more difficult with explicit threads.
- Implicit threading: creation and management of threads done by compilers and run-time libraries rather than programmers.
- Four methods explored:
  - Thread Pools
  - Fork-Join
  - OpenMP
  - Grand Central Dispatch

Thread Pools

- Create a number of threads in a pool where they await work.
- Usually slightly faster to service a request with an existing thread than create a new thread.
- Allows the number of threads in the application(s) to be bound to the size of the pool.

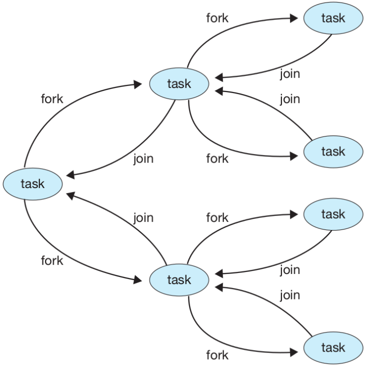
Fork-Join (1/2)

- Multiple threads (tasks) are forked, and then joined.



Fork-Join (2/2)

```
Task(problem)
  if problem is small enough
    solve the problem directly
  else
    subtask1 = fork(new Task(subset of problem))
    subtask2 = fork(new Task(subset of problem))

    result1 = join(subtask1)
    result2 = join(subtask2)

    return combined results
```



OpenMP (1/2)

- Set of compiler directives and APIs for C, C++, FORTRAN.
- Identifies parallel regions: blocks of code that can run in parallel.
- `#pragma omp parallel`: create as many threads as there are cores.
- `#pragma omp parallel for`: run for loop in parallel.

OpenMP (2/2)

```c
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[]) {

  /* sequential code */

  #pragma omp parallel
  {
    printf("I am a parallel region.");
  }

  /* sequential code */

  return 0;
}
```

Grand Central Dispatch

- Apple technology for Mac OS X and iOS: extensions to C, C++ API, and run-time library.
- Allows identification of parallel sections.
- Block is in `^{ }`: `^{ printf("I am a block"); }`
- Blocks placed in dispatch queue.

```
dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH QUEUE PRIORITY DEFAULT, 0);

dispatch_async(queue, ^{ printf("I am a block."); });
```

Threading Issues

Threading Issues

- The `fork()` and `exec()` system calls
- Signal handling
- Thread-Local Storage (TLS)
- Thread cancellation

The `fork()` and `exec()` System Calls

- Does `fork()` duplicate only the calling thread or all threads?
  - Some UNIXes have two versions of fork.
- `exec()` usually works as normal: replace the entire process, including all threads.

Signal Handling (1/2)

- Signals are used in UNIX systems to notify a process that a particular event has occurred.
- A signal handler is used to process signals:
  1. Signal is generated by particular event.
  2. Signal is delivered to a process.
  3. Signal is handled by the signal handlers, either the default or user-defined.
- Where should a signal be delivered for multi-threaded?

Signal Handling (2/2)

- Where should a signal be delivered for multi-threaded?
  - Deliver the signal to the thread to which the signal applies.
  - Deliver the signal to every thread in the process.
  - Deliver the signal to certain threads in the process.
  - Assign a specific thread to receive all signals for the process.

Thread-Local Storage (TLS)

- TLS allows each thread to have its own copy of data.
- Useful when you do not have control over the thread creation process (i.e., thread pool)
- Different from local variables:
  - Local variables visible only during single function invocation.
  - TLS visible across function invocations.

Thread Cancellation (1/4)

- Terminating a thread before it has finished.
- Thread to be canceled is target thread.
- Two general approaches:
  - Asynchronous cancellation terminates the target thread immediately.
  - Deferred cancellation allows the target thread to periodically check if it should be cancelled.

Thread Cancellation (2/4)

```
int counter = 0;

pthread_t tmp_thread;

void* thread_func1(void* args) {
  while (1) {
    printf("thread number one\n");
    sleep(1);
    counter++;

    if (counter == 2) {
      pthread_cancel(tmp_thread);
      pthread_exit(NULL);
    }
  }
}
```

Thread Cancellation (3/4)

```
void* thread_func2(void* args) {
  tmp_thread = pthread_self();

  while (1) {
    printf("thread number two\n");
    sleep(1); // sleep 1 second
  }
}
```

Thread Cancellation (4/4)

```
int main() {
  pthread_t thread1, thread2;

  pthread_create(&thread1, NULL, thread_func1, NULL);
  pthread_create(&thread2, NULL, thread_func2, NULL);

  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
}
```

Pthread Hands-On 3

```c
struct thread_args {
  int a;
  double b;
};


struct thread_result {
  long x;
  double y;
};

void *thread_func(void *args_void) {
  struct thread_args *args = args_void;
  struct thread_result *res = malloc(sizeof *res);
  res->x = args->a * 2;
  res->y = args->b * 2;
  return res;
}

int main() {
  pthread_t thread;
  struct thread_args in = { .a = 10, .b = 3.14 };
  void *out_void;
  struct thread_result *out;

  <YOUR CODE>
}
```

Summary

Summary

- Single-thread vs. Multi-thread
- Concurrency vs. parallelism
- Multi-threading models: many-to-one, one-to-one, many-to-many
- Multi-thread libraries: pthread
- Implicit threading
- Threading issues

Questions?

Acknowledgements

Some slides were derived from Avi Silberschatz slides.