System Programming

6. Thread Programming

Seung-Ho Lim

Dept. of Computer & Electronic Systems Eng.

Concurrent Processes

- Processes are heavy-weight
  - A process includes many things:
    - An address space (all the code and data pages)
    - OS resources (e.g., open files) and accounting info.
    - Hardware execution state (PC, SP, registers, etc.)
  - Creating a new process is costly because all of the data structures must be allocated and initialized
    - Linux: over 100 fields in task_struct (excluding page tables, etc.)
  - Inter-process communication is costly, since it must usually go through the OS
    - Overhead of system calls and copying data

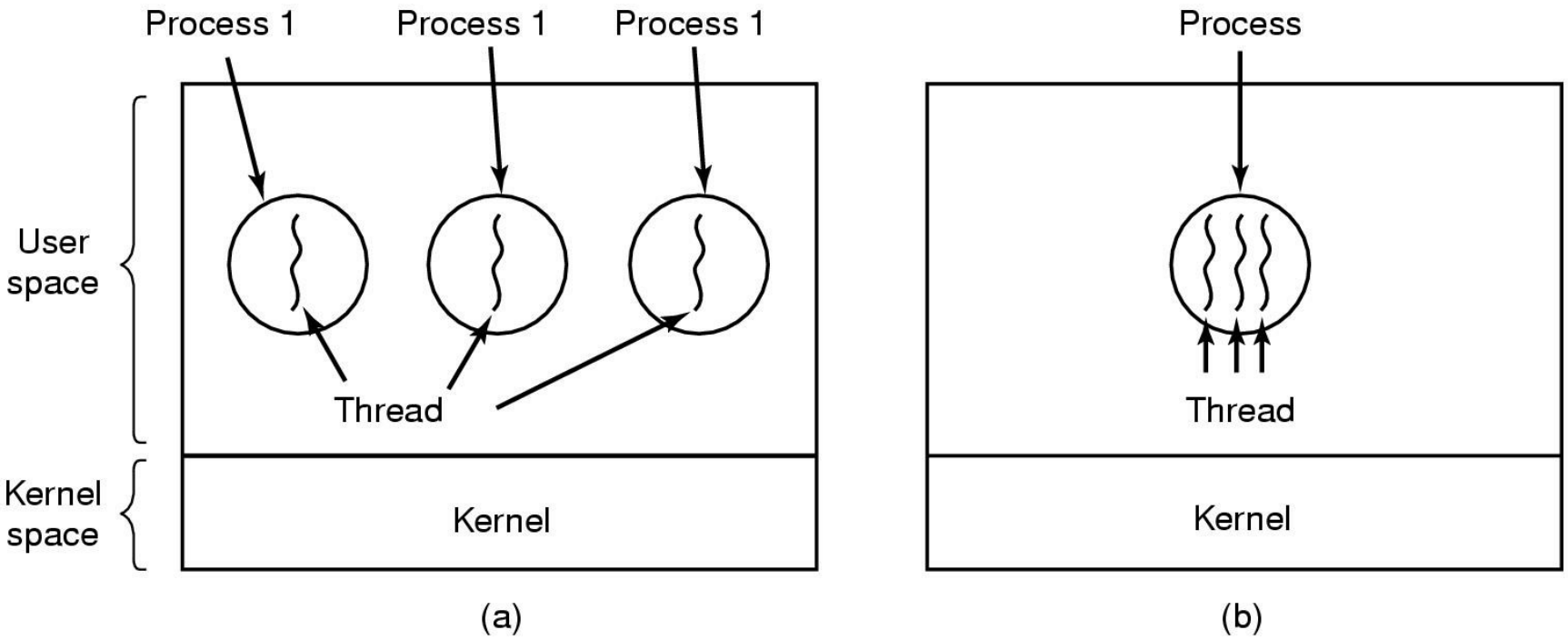
Rethinking the processes?

- What's similar in these cooperating processes?
  - They all share the same code and data (address space)
  - They all share the same privilege
  - They all share the same resources (files, sockets, etc.)
- What's different?
  - Each has its own hardware execution state: PC, registers, SP, and stack.
- Key idea
  - Separate the concept of a process from its execution state

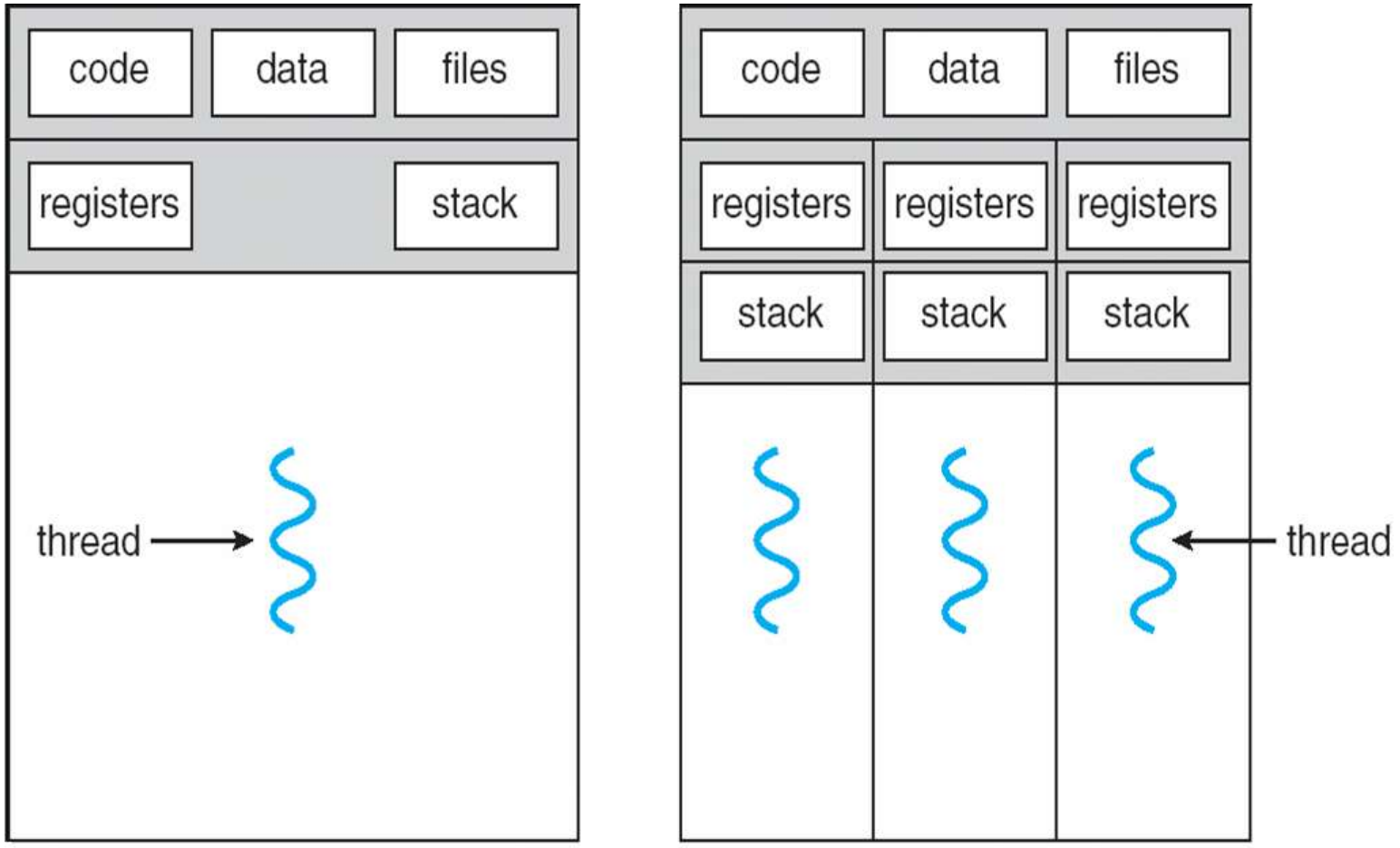
What is a thread? (1)

- A thread of control (or just called a thread)
  - a flow of control within a process
  - a process is also single-threaded process
  - multi-threaded process has a multiple flow control within a process
  - a thread is a CPU scheduling unit
- A thread consist of
  - thread ID, a program counter, registers
  - stack to keep track of local variables and return address
- Multiple threads in a process share
  - code section, data section, OS resources (e.g. open files, signals)
  - A change in shared data by one thread can be seen by the other threads in the process
  - the threads also share most of the OS state of the process

What is a thread? (2)



Single- vs. Multiple-threaded process



single-threaded process

multithreaded process

Multi-threaded applications (1)

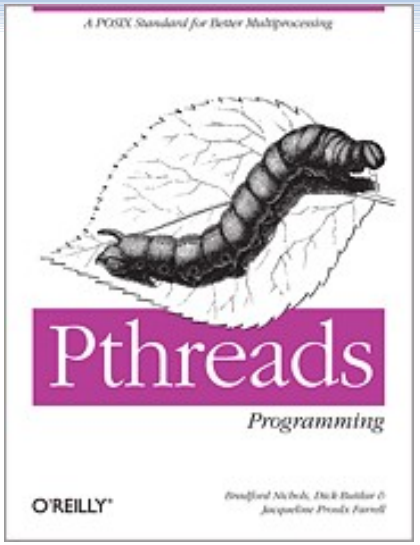
- Most of packages in modern OS supports multi-threading
  - MT(multi-threaded)-safe library functions can be used in multi-threading purpose
- Cases
  - web browser
    - a thread which display images and text
    - another thread which receives data from network
  - word processor
    - a thread which draws a graph
    - second thread which reads key or mouse events by user
    - third thread which check spelling and grammar

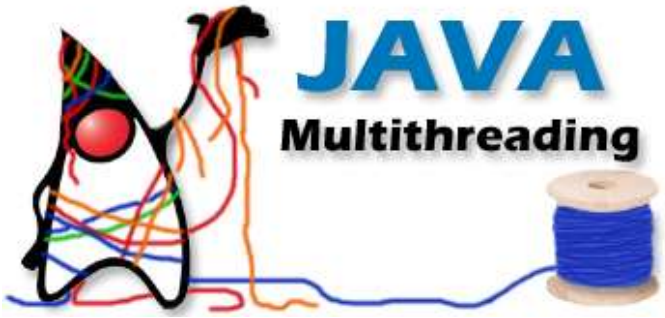
Multi-threaded applications (2)

- Web server case
  - single-threaded server
    - requests from users are handled sequentially by one process
  - multi-process server
    - a new process is created for a new client
    - high process overhead
  - multi-threaded server
    - almost same as the multi-process server, but the overhead is quite small
- Most OS kernels are based on multi-threaded architecture
  - multiple threads in a kernel
  - each thread performs its own task to manage a device or handle an interrupt

Thread libraries

- Linux/Unix
  - POSIX Pthreads
  - Mach C-threads, Solaris Threads
- Windows
  - Win32 Threads
- JVM
  - Java Threads (Thread class, Runnable interface)





POSIX Pthreads

- POSIX standard API (IEEE 1003.1c)
  - thread creation, management, and synchronization
  - just API specification about behavior of the threads
  - implementation is up to development of the library
    - may be different from OS to OS
  - common in Linux/Unix operating systems
- pthreads
  - defined in `<pthread.h>` header file
  - implemented in `libpthread.a` library
  - compile option: -lpthread

```
$ gcc test.c -lpthread
```

Pthreads API and data types

prefix	functions
`pthread_`	Threads themselves and miscellaneous subroutines
`pthread_attr_`	Thread attributes objects
`pthread_mutex_`	Mutex routines
`pthread_mutexattr_`	Mutex attribute objects
`pthread_cond_`	Condition variable routines
`pthread_condattr_`	Condition attribute objects
`pthread_key_`	Thread_specific_data keys

pthread_create (1)

```
#include <pthread.h>

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

- a function which creates a (new) thread
- parameters (all parameters are passed by reference (void *))
  - *thread*: pointer variable to store a thread ID returned
  - *attr*: thread attributes for the new thread (`default: NULL`)
  - *start_routine*: a function performed by the new thread
  - *arg*: argument passed to the thread function(i.e. *start_routine*)
- return
  - 0 if Ok
  - error number on an error

pthread_create (2)

- Note
  - initially, your `main()` program comprises a single, default thread
  - all other threads must be explicitly created by the programmer
  - once a thread is created, all threads are peers
  - the maximum number of threads in a process is implementation dependent.
    - cat /proc/sys/kernel/threads-max (default: 388547)

Terminating a pthread (1)

- There are several ways in which a pthread may be terminated:
  - a thread returns from its start routine
  - a thread makes a call to `pthread_exit` function
  - the thread is cancelled by another thread via the `pthread_cancel` function (*will see later*)
  - the entire process is terminated due to exit call

Terminating a pthread (2)

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

- parameter
  - *retval*: a return value delivered to another thread that calls `pthread_join` to wait for this thread to join
  - type is cast to (void *) to support general data types
- note
  - Recommend to use `pthread_exit` in all threads (especially, main)
    - If main() finishes with pthread_exit(), the other threads will continue to execute.
    - Otherwise, they will be automatically terminated when main() finishes exit call
  - the `pthread_exit()` routine does not close files; any files opened inside the thread will remain open after the thread is terminated (when cleanup?)

Detaching/Joining a thread (1)

- When a thread is created, one of its attributes defines whether it is joinable or detached.
  - ***detached*** thread

```
int pthread_detach(pthread_t id);
```

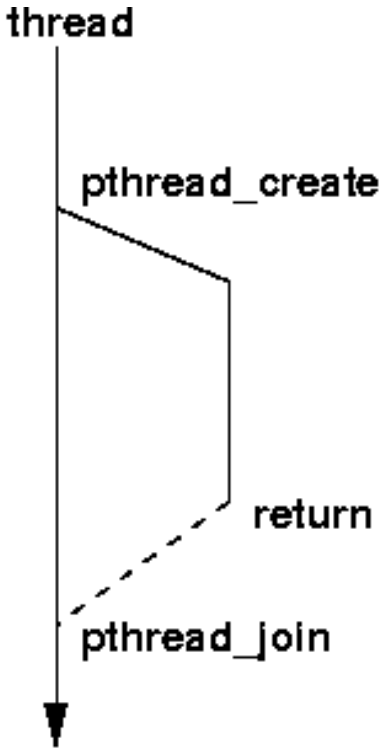
    - it can never be joined, and independently managed
    - once detached, the thread cannot join again
    - when a detached thread finishes, the resources are immediately claimed.
  - ***joinable*** thread
    - it can join to another thread
    - when the joinable thread finishes (exits)
      - » the resources used by the thread are not claimed until it is joined by another thread.

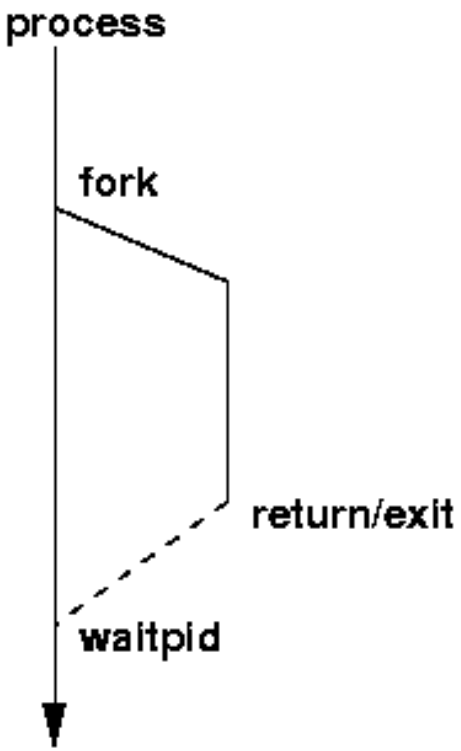
Detaching/Joining a thread (2)

```
int pthread_join(pthread_t tid, void **retval);
```

- parameters
  - *tid* : a thread id to wait for joining
  - *retval*: a pointer to a return value from the target thread
- return
  - 0 if Ok, or an error number on error
- note
  - The ***pthread_join()*** routine blocks the calling thread until the specified thread (*thread_id*) terminates.
  - The programmer is able to obtain the target thread's termination status if specified through ***pthread_exit(****retval****),*** in return value
    - the return value is also used to indicate a status
  - if the target thread is cancelled, **PTHREAD_CANCELED** is placed on *retval* variable.

pthread join vs. process wait





Getting Thread ID

```
int pthread_self(void);
```

- returns a thread id of the calling thread

```
int pthread_equal(pthread_t t1, pthread_t t2);
```

- returns
  - a nonzero if the two thread IDs are equal
  - 0 otherwise
- note
  - the C language equivalence operator == should not be used to compare two thread IDs against each other.

pthread example (1)

th_hello.c

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

int global;

void *printmsg(void *msg)
{
    int i=0;

    while (i<5) {
        printf("%s : %d \n", (char *) msg, i++);
        sleep(1);
    }

    pthread_exit((void *)pthread_self());
}
```

pthread example (2)

```
void main()
{
    pthread_t   t1, t2, t3;
    void *rval1;
    void *rval2;
    int mydata;

    pthread_create(&t1, NULL, printmsg, "Hello");
    pthread_create(&t2, NULL, printmsg, "World");

    pthread_join(t1, (void *)&rval1);
    pthread_join(t2, (void *)&rval2);

    printf("t1: %lu, t2: %lu \n", t1, t2);
    printf("rval1: %lu, rval2: %lu \n",
       (unsigned long)rval1, (unsigned long)rval2);

}
```

pthread example (3)

```
# gcc –o th_hello th_hello.c -lpthread
#./th_hello
World : 0
Hello : 0
World : 1
Hello : 1
World : 2
Hello : 2
World : 3
Hello : 3
World : 4
Hello : 4
t1: 139882789895993, t2: 139887043430144
rval1: 139882789895993, rval2: 139887043430144
```

Thread cancellation (1)

- Thread cancellation
  - terminating a thread before it has completed
  - call `pthread_cancel(pthread_t tid)`
- Cancellation types
  - asynchronous cancellation
    - terminates the target thread immediately.
    - if the target thread is holding a resource, or it is in the middle of updating shared resources?
  - deferred cancellation
    - the thread cancellation is deferred (not immediately)
    - the target thread is terminated at the cancellation points.
    - the target thread periodically check if it should be cancelled
    - call `pthread_testcancel()` to make a cancellation point
    - call `pthread_cleanup_push()` to register a cleanup handler which is called at cancellation point

Thread cancellation (2)

- Functions related to cancellation

```
int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
```

- Cancellation modes (PTHREAD_CANCEL_??)

mode	state	type
_DISABLE	NOT cancellable	
_ENABLE	cancellable	not used
_DEFERRED	cancellable	deferred
_ASYNCHRONOUS	cancellable	immediate

Thread cancel example (1)

th_cancel.c

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

void *threadFunc(void *arg)
{
    int count = 0;
    printf("new thread started ....\n");
    int retval;

    pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
    pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
    while(1)
    {
        printf("count = %d \n", count++);
        sleep(1);
        pthread_testcancel();

        // for testing pthread_exit() return value
        if(count == 10) break;
    }
    pthread_exit((void *) count);
}
```

Thread cancel example (2)

th_cancel.c

```
int main(int argc, char *argv[])
{
    pthread_t tid;
    int retval;
    void *res;

    pthread_create(&tid, NULL, threadFunc, NULL);
    sleep(5);
    pthread_cancel(tid);
    retval = pthread_join(tid, &res);
    if(retval != 0)
    {   perror("pthread_join : ");
        exit(EXIT_FAILURE);
    }
    if(res == PTHREAD_CANCELED)
        printf("thread canceled\n");
    else
        printf("thread is normal exit retval = %d \n",(int)res);

    exit(EXIT_SUCCESS);
}
```

Thread cancel example (3)

th_cancel.c

```
#./th_cancel
new thread started ....
count = 0
count = 1
count = 2
count = 3
count = 4
count = 5
count = 6
count = 7
count = 8
count = 9
Thread is normal exit retval = 10
```

Thread arguments example (1)

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

#define NUM_THREADS3

struct thread_data {
        int thread_id;
        int sum;
        char *message;
};

struct thread_data   thread_data_array[NUM_THREADS];
pthread_t threads[NUM_THREADS];
```

Thread arguments example (2)

```
void *printHello(void *arg)
{
        struct thread_data *my_data;
        int taskid, sum;
        char *hello_msg;

        my_data = (struct thread_data *) arg;
        taskid = my_data->thread_id;
        sum = my_data->sum;
        hello_msg = my_data->message;

        printf("taskid = %d\n",taskid);
        printf("sum = %d\n", sum);
        printf("message = %s\n",hello_msg);

}
```

Thread arguments example (3)

```
int main (int argc, char *argv[])
{
        int rc, I, sum;
        void *res;
        char messages[3][1024];
        strcpy(messages[0], "hello");
        strcpy(messages[1], "system programming");
        strcpy(messages[2], "world");

        for(i=0;i<3;i++) {
            sum += i;
            thread_data_array[i].thread_id = i;
            thread_data_array[i].sum = sum;
            thread_data_array[i].message = messages[i];
            rc = pthread_create(&threads[i], NULL, printHello, (void *) &thread_data_array[i]);
        }
        for(i=0;i<3;i++) {
            pthread_join(threads[i], &res);
        }
}
```

Thread arguments example (4)

```
#./th_argument
taskid = 2
sum = 3
message = world
taskid = 1
sum = 1
message = system programming
taskid = 0
sum = 0
message = hello
```

Dangerous Argument Passing

- What will happen in this program?

```
int rc, i;

for(i=0; i < NUM_THREADS;i++) {
   printf("Creating thread %d\n", i);
   rc = pthread_create(&threads[i], NULL, routine, (void *) &i);
   ...
}
```

- This example shows **implicit sharing** b/w threads
  - contents of the address passed as an argument may be changed after delivering a value to a thread
  - the variable becomes shared between threads by mistake
  - such implicit sharing may frequently occur

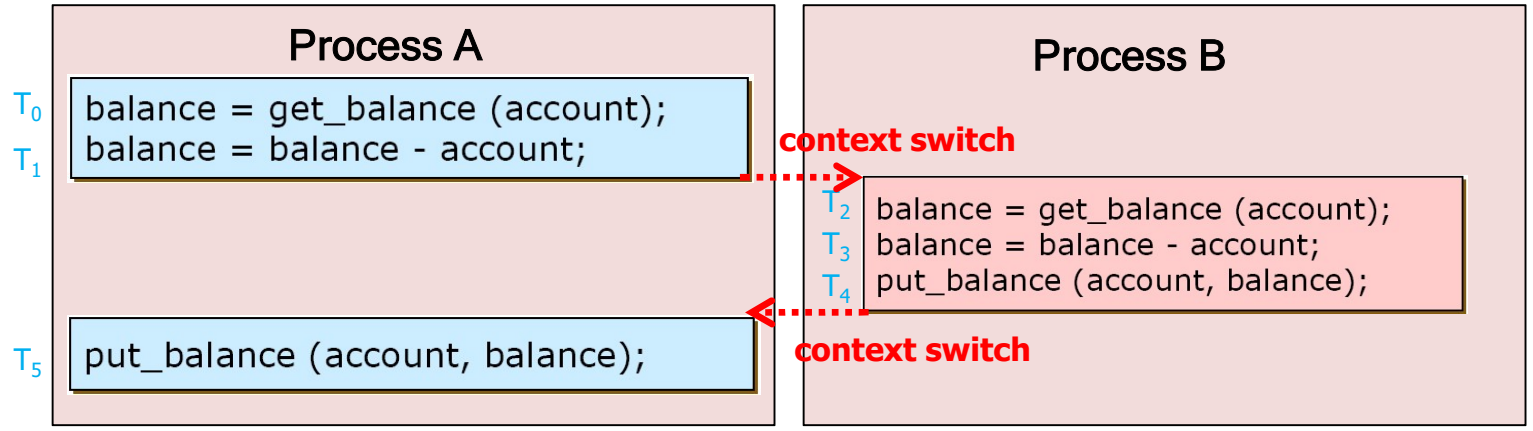
Thread Synchronization (1)

- Let's think of this example
  - Suppose you and your girl(boy) friend share a bank account with a balance of 1,000,000won.
  - What happens if both of you go to separate ATM machines, and simultaneously withdraw 100,000won from the account?

```
int withdraw (account, amount)
{
  balance = get_balance (account);
  balance = balance - amount;
  put_balance (account, balance);
  return balance;
}
```

Thread Synchronization (2)

- Threads execution and sharing
  - In this case, the banking system will create two threads (or processes), which share the variable balance.
  - The execution of the two processes can be interleaved, assuming preemptive scheduling:
  - What value will the balance hold as a result?



Mutex (mutual exclusion) (1)

- **Mutex** variables
  - one of the primary means of implementing thread synchronization and for protecting shared data when multiple writes occur.
  - A **mutex** variable acts like a "**lock**" protecting access to a shared data resource.
- The basic concept of a **mutex** as used in Pthreads
  - only one thread can **lock (acquire) a mutex** variable at any given time.
  - thus, even if several threads try to **lock (acquire) a mutex** only one thread will be successful.
  - no other thread can own that **mutex** until the lock owner thread **unlocks (release) the mutex**.
    - threads must "take turns" accessing protected data.

Mutex (2)

- In the banking example,
  - balance is a shared variable (resource)
- Critical section
  - a code section that read and writes a shared variable
  - which part is a critical section in the banking example?
- A critical section should be mutually exclusive (with other processes/threads)
  - critical sections must be protected by a mutual exclusion

pthread mutex creation (1)

```
/* mutex object creation & destruction */
int pthread_mutex_init (*mutex, *attr);
int pthread_mutex_destroy (*mutex);
/* mutex attribute creation & destruction */
int pthread_mutexattr_init (*attr);
int pthread_mutexattr_destroy (*attr);

pthread_mutex_t *mutex
pthread_mutexattr_t *attr
```

- Mutex variable must be initialized after declaration
- Mutex variable: two methods for initialization
  - statically, declare :
    `pthread_mutex_t mymutex = THREAD_MUTEX_INITIALIZER;`
  - at run time, call the `pthread_mutex_init(attr)`
- The mutex is initially unlocked

pthread_mutexattr

- pthread_mutexattr_t defines the
  - protocol
    - specifies the protocol used to prevent priority inversions for a mutex.
  - prioceiling
    - specifies the priority ceiling of a mutex.
  - process-shared
    - specifies a mutex which is shared b/w processes
- set NULL to use default attributes

Mutex Lock/Unlock (1)

```
int pthread_mutex_lock(*mutex);
int pthread_mutex_trylock(*mutex);
int pthread_mutexattr_unlock(*mutex);

pthread_mutex_t *mutex
pthread_mutexattr_t *attr
```

- returns
  - 0 if successful, or error number on error
- pthread_mutex_lock(*mutex)
  - used by a thread to lock on the specified mutex variable
  - If the mutex is already locked by another thread, this call will block the calling thread until the mutex is unlocked.

Mutex Lock/Unlock (2)

- pthread_mutex_trylock(*mutex)
  - unblocking try for locking a mutex
- pthread_mutex_unlock(*mutex)
  - unlock a mutex by the lock holding thread
  - an error will be returned if
    - if the mutex was already unlocked
    - if the mutex is held by another thread

- Advisory locking by programmer (not by system)
  - error-prone example

Thread 1	Thread 2	Thread 3
Lock	Lock	
A = 2	A = A+1	A = A*B
Unlock	Unlock	

Simple Mutex example (1)

mutex_counter.c

```
void *Incrementer();
void *Decrementer();
int counter = 0 ;

pthread_mutex_t mVar = PTHREAD_MUTEX_INITIALIZER;

int main()
{  pthread_t  ptid, ctid;

   pthread_mutex_init(&mVar, NULL) ;

   pthread_create(&ptid,NULL,Incrementer,NULL);
   pthread_create(&ctid,NULL,Decrementer,NULL);
   pthread_join(ptid,NULL);
   pthread_join(ctid,NULL);
   return 0;
}
```

Simple Mutex example (2)

mutex_counter.c

```
void *Incrementer()
{   for(;;) {
        pthread_mutex_lock(&mVar);
        counter++ ;
        printf("Inc : %d \n", counter);
        pthread_mutex_unlock(&mVar);
    }
}

void *Decrementer()
{   for(;;) {
        pthread_mutex_lock(&mVar);
        counter-- ;
        printf("Dec : %d \n", counter);
        pthread_mutex_unlock(&mVar);
    }
}
```

Multi-threaded Dot-Product Example (1)

- A **dot product** program
  - compute two big vector product of A[] and B[]
    - for all i, sum += A[i]*B[i]
  - Each thread works on a part of two big vector
    - for j in range of each thread, local_sum += A[j]*B[j]
  - Main thread waits for all the threads to complete their computations, and then it prints the resulting sum.
    - from each thread, get its local_sum
    - sum += local_sum of the thread

Multi-threaded Dot-Product Example (2)

dot_product.c

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

typedef struct {
   double *a;  // first vector
   double *b;  // second vector
   double sum; // dot product of two vectors
   int veclen; // dimension
} DOTDATA;

#define NUMTHRDS 4
#define VECLEN 100

DOTDATA dotstr;

pthread_t callThd[NUMTHRDS];
pthread_mutex_t mutexsum;
```

Multi-threaded Dot-Product Example (3)

```
void *dotprod(void *arg) {
   int i, start, end, offset, len ;
   double mysum, *x, *y;

   offset = (int)arg;
   len = dotstr.veclen;
   start = offset*len;
   end = start + len;
   x = dotstr.a;   y = dotstr.b;

   /* Perform the dot product */
   mysum = 0;
   for (i=start; i < end ; i++) {
        mysum += (x[i] * y[i]);
   }
   /* Lock a mutex prior to updating the value in the shared structure,
     and unlock it upon updating. */
   pthread_mutex_lock (&mutexsum);
   dotstr.sum += mysum;
   pthread_mutex_unlock (&mutexsum); pthread_exit((void*) 0);
}
```

Multi-threaded Dot-Product Example (4)

```
int main (int argc, char *argv[]) {
   int i;
   double *a, *b;
   int status;


   a = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));
   b = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));

   for (i=0; i < VECLEN*NUMTHRDS; i++)
        b[i]= a[i]=1; // for easy testing


   dotstr.veclen = VECLEN;
   dotstr.a = a;
   dotstr.b = b;
   dotstr.sum=0;
   pthread_mutex_init(&mutexsum, NULL);
```

Multi-threaded Dot-Product Example (5)

```
        /* Create threads to perform the dot-product */
        for(i=0;i < NUMTHRDS; i++) {
              pthread_create( &callThd[i], NULL,
                                    dotprod, (void *)i);
        }

        /* Wait on the other threads */
        for(i=0;i < NUMTHRDS; i++) {
              pthread_join( callThd[i], (void **)&status);
        }

        /* After joining, print out the results and cleanup */
        printf ("Sum = %f \n", dotstr.sum);
        free (a); free (b);

        pthread_mutex_destroy(&mutexsum);
        pthread_exit(NULL);
}
```