Process Programming

What is process?

Each logical control flow is a process that is scheduled and maintained by the kernel;

Processes have separae virtual address spaces;

Flows use interprocess communication (IPC) to communicate with each other;

Pros: It is impossible for one process to accidentally overwrite the virtual memory of another process, which eliminates a lot of confusing failures

Cons: Information share involves IPC, the overhead of which is high

Create a process via pid_t fork(void)

Fork() 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, except for the following points:

- The child has its own unique process ID, and this PID does not match the ID of any existing process group

- The child does not inherit its parent's memory locks

- The child's set of pending signals is initially empty

- The child does not inherit semaphore adjustments from its parent

Return value of fork()

On success, the PID of the child process is returned in the parent, and 0 is Returned in the child. On failure, -1 is returned in the parent, no child process is created, and errno is set appropriately.

Process wait

Source: https://linux.die.net/man/2/waitpid

```
pid_t wait(int *status);

pid_t waitpid(pid_t pid, int *status, int options);

int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

All of these system calls are used to **wait for state changes in a child** of the calling process, and obtain information about the child whose state has changed.

A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal. In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then the terminated child remains in a "zombie" state

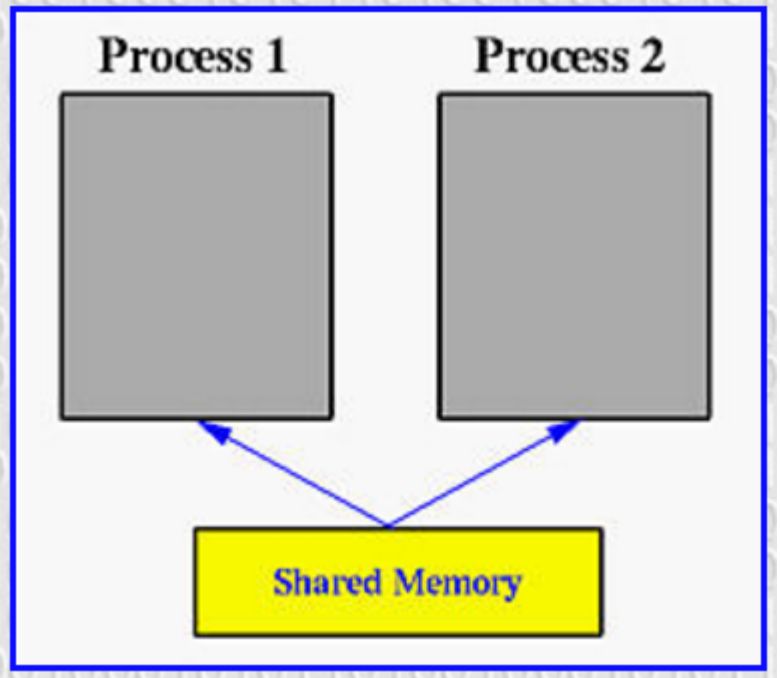
https://linux.die.net/man/2/waitpid

Code Demonstration

IPC via Shared Memory

http://www.csl.mtu.edu/cs4411.ck/www/NOTES/process/shm/shmat.html

Suppose process 1 and process 2 have successfully attached the shared memory segment. This shared memory segment will be part of their address space, although the actual address could be different (i.e., **the starting address of this shared memory segment in the address space of process 1 may be different from the starting address in the address space of process 2**).



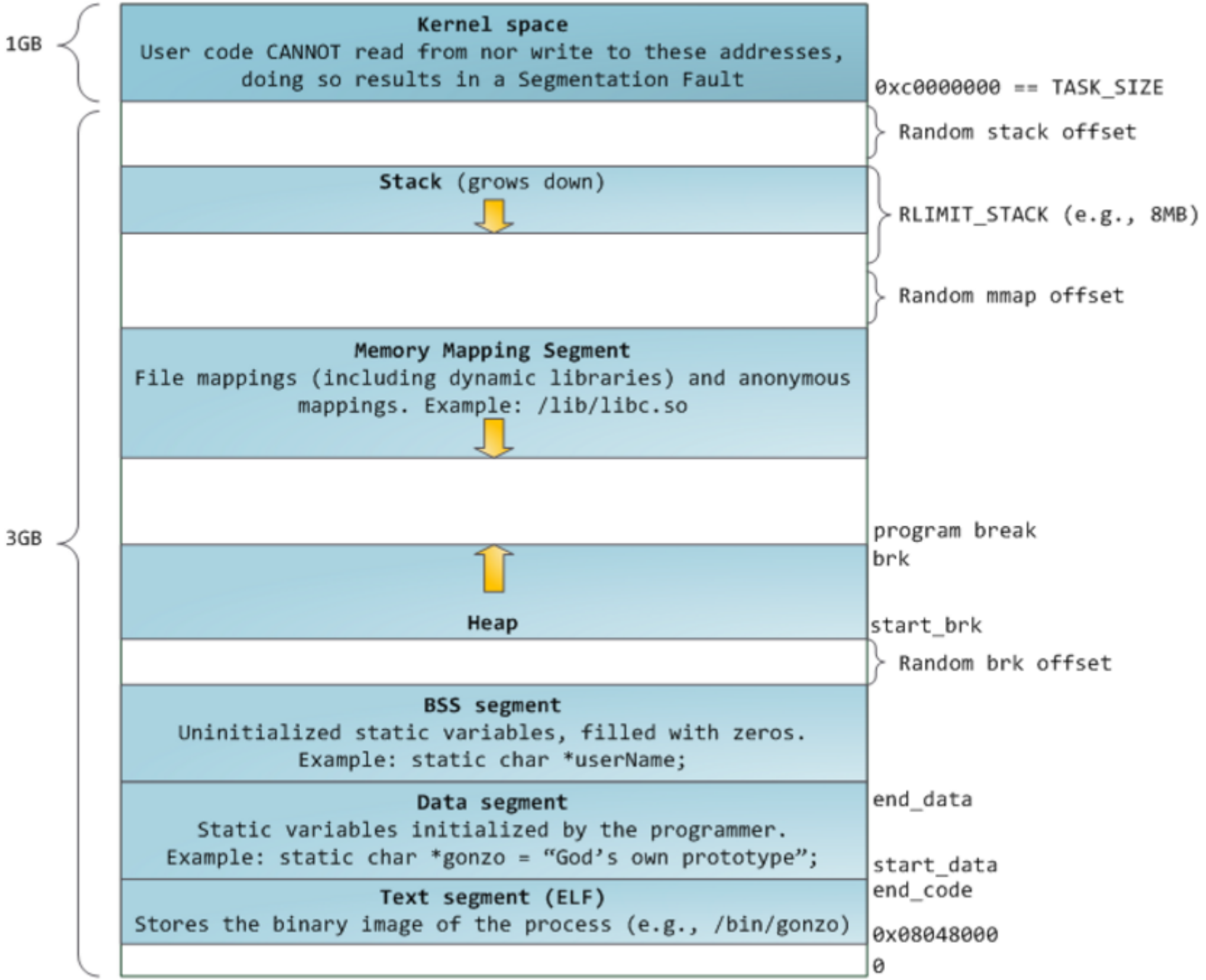
```
shared_memory = (char*) shmat ( segment_id,
                                //specify the identifier of shared memory segment


                               (void*) 0x5000000,
                                // a pointer that specifies where in your process's address
                                // space you want to map the shared memory; if you specify
                                // NULL, Linux will choose an available address.

                                Should not overwrite the original address space


                                0);
                                //0, it is readable and writable.
```

http://stackoverflow.com/questions/29580251/understand-shmat-and-attachment-to-the-process-memory



Another function approaching shared memory

http://man7.org/linux/man-pages/man2/mmap.2.html

void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);

mmap() creates a new mapping in the virtual address space of the calling process.

The starting address for the new mapping is specified in addr. The length argument specifies the length of the mapping.

If addr is NULL, then the kernel chooses the address at which to create the mapping; this is the most portable method of creating a new mapping.

If addr is not NULL, then the kernel takes it as a hint about where to place the mapping; on Linux, the mapping will be created at a nearby page boundary.

The address of the new mapping is returned as the result of the call.

Another function approaching shared memory

http://man7.org/linux/man-pages/man2/mmap.2.html

The flags argument determines whether updates to the mapping are visible to other processes mapping the same region, and whether updates are carried through to the underlying file. This behavior is determined by including exactly one of the following values in flags:

MAP_SHARED

Share this mapping. Updates to the mapping are visible to other processes mapping the same region, and (in the case of file-backed mappings) are carried through to the underlying file. (To precisely control when updates are carried through to the underlying file requires the use of msync(2).)

Multi-threaded programming

Source: computer systems, a programmer's perspective. 3rd edition

What are threads:

Threads are scheduled automatically by the kernel

`The stack segment is divided among threads.`

Each thread has its own thread context, including a unique integer thread ID (TID), stack, stack pointer, program counter, general-purpose registers and condition codes.

`Threads share heap, global variable (on uninitialized memory segment or initialized memory segment)`

Relationship among threads:

The threads associated with a process form a pool of peers, independent of which threads were created by which other threads:

(1) the main thread is distinguished from other threads only in the sense that it is always the first thread to run in the process;

(2) a thread can kill any of its peers or wait for any of its peers to terminate;

(3) each peer can read and write the same shared data

Some useful functions

```
#include “csapp.h”
void* thread(void* vargp)

int main( )
{
   pthread_t  tid;
   Pthread_create(&tid, NULL, thread, NULL);
   Pthread_join(tid, NULL);
   exit(0);
}
```

When this call returns, main thread and peer thread are running concurrently

Wait for other threads to terminate. It blocks until thread tid terminates. Unlike Linux wait function, Pthread_join this function can only wait for a specific thread to terminate

```
void* thread(void* vargp)  /*thread routine*/
{
   prhead_t selftid = pthread_self();
   printf(“Hello, world\n”);
   return NULL;
}
```

The new thread can determine its own thread ID by calling the pthread_self function

Create threads

int pthread_creat(pthread_t* tid, pthread_attr_t* attr, void * (*start_routine)(void *), void* arg);

This guy always NULL

Thread prototype

If you want to pass multiple arguments to a thread routine, you should put the arguments into a structure and pass a pointer to the structure

Similarly, if you want the thread routine to return multiple arguments, You can return a pointer to a structure

Terminate threads

Self termination:

```
#include<pthread.h>
void pthread_exit(void* thread_return);
```

return value from a thread

If the main thread calls pthread_exit, it waits for all other peer threads to terminate and then terminates the main thread and the entire process with a return value of thread_return

Terminate a peer thread:

```
#include<pthread.h>
void pthread_cancel(pthread_t tid);
```

The pthread_cancel() function sends a cancellation request to the thread thread. Whether and when the target thread reacts to the cancellation request depends on two attributes that are under the control of that thread: its cancelability state and type.

http://man7.org/linux/man-pages/man3/pthread_cancel.3.html

Wait a thread to terminate and receive the thread's return value

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

Accept the return value from the thread tid

This implies, if thread A wants to receive the return of thread B,
A has to wait for B to terminate via pthread_join(void thread_return)**

About return a value from thread, check this site:
http://stackoverflow.com/questions/2251452/how-to-return-a-value-from-thread-in-c

"If you need to return a complicated value such a structure, it's probably easiest to allocate it dynamically via malloc() and return a pointer. Of course, the code that initiated the thread will then be responsible for freeing the memory."

Detach threads

A thread is either joinable (ie., waitable, by default) or detached.

A detached thread can not be waited or terminated by other threads.

```
#include<pthread.h>
int pthread_detach(pthread_t tid);
```

Initializing thread

```
#include<pthread.h>
pthread_once_t once_control = PTHREAD_ONCE_INIT;

Int pthread_once(pthread_once_t* once_control,
                 void (*init_routine)(void) );
```

once_control variable is always initialized to PTHREAD_ONCE_INIT ==> subsequent calls to pthread_once( ) with the same once_control variable do nothing

A function with no input arguments that returns nothing

Mutexes of threads

http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html

Purpose of Mutexes (a resource object about data access permission)

- Mutexes are used to prevent data inconsistencies due to operations by multiple threads upon the same memory area performed at the same time or to prevent race conditions where an order of operation upon the memory is expected.
- Mutexes can be applied only to threads in a single process and do not work between processes as do semaphores.
- A mutex is an object that has two possible states: unlocked (not owned by any thread), and locked (owned by one thread). A mutex can never be owned by two different threads simultaneously.

Three key functions

pthread_mutex_lock()
acquire a 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.

pthread_mutex_trylock() (Lock Mutex with No Wait)
Attempt to lock a mutex or will return error code if the mutex is already locked by another thread. Useful for preventing deadlock conditions.

pthread_mutex_unlock()
Unlock a mutex variable. An error is returned if mutex is already unlocked or owned by another thread.
Details for this unlock function, see
http://www.agr.unideb.hu/~agocs/informatics/11_e_unix/unixhelp/unixhelp.ed
.ac.uk/CGI/man-cgide5a.html?pthread_mutex_unlock+3

Code Example

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

void *functionC();
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;  //mutex1 is fast type
int  counter = 0;    //shared variable

Main() {
   pthread_t thread1, thread2;

   /* Create independent threads each of which will execute functionC */
   pthread_create( &thread1, NULL, &functionC, NULL);
   pthread_create( &thread2, NULL, &functionC, NULL);

   /* Wait till threads are complete before main continues.*/
   pthread_join( thread1, NULL);
   pthread_join( thread2, NULL);

   exit(EXIT_SUCCESS);
}

void *functionC() {       //thread function
   pthread_mutex_lock( &mutex1 );
   counter++;
   printf("Counter value: %d\n",counter);
   pthread_mutex_unlock( &mutex1 );  //When a thread terminates, the mutex does
                                     //not unless explicitly unlocked. Nothing happens by default.
}
```

http://www.2net.co.uk/tutorial/mutex_mutandis

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER; **//mutex1 is fast type**

Linux has four types of mutex. Default is fast type.
Fast type means that speed is preferred over correctness: there is no check that you are the owner in pthread_mutex_unlock() so any thread can unlock a fast mutex. Also it doesn't check if you have already locked the mutex, so you can deadlock yourself, and there are no checks anywhere that the mutex has been intialised correctly.

Compile: cc -pthread mutex1.c (or cc -lpthread mutex1.c for older versions of the GNU compiler which explicitly reference the library)
Run: ./a.out
Results:

Counter value: 1
Counter value: 2

Q: what might happen if there is no mutex?

Counter ==> temp ==> temp+1 ==> Counter

Counter value: 1	Counter value: 2
Counter value: 1	Counter value: 2