OPERATING SYSTEMS

Threads

Operating Systems

School of Data & Computer Science
Sun Yat-sen University

Lecture Notes: os_sysu@163.com
Instructor: Guoyang Cai
email: isscgy@mail.sysu.edu.cn

中山大學
SUN YAT-SEN UNIVERSITY

Contents

Thread Local Storage

- Thread-local storage (TLS) allows each thread to have its own copy of data.
- TLS is useful when we do not have control over the thread creation process.
  - We can not pass any parameters to the created thread.
  - e.g., when using a thread pool.
- TLS is different from local variables.
  - Local variables are visible only during single function invocation.
  - TLS is visible across function invocations.
- Similar to *static data*:
  - TLS is unique to each thread.
- Implementation of TLS
  - __thread int tlsvar; /* tlsvar for each thread; interpreted by language compiler, a language level solution to TLS */
  - by pthread_key_create

Thread Local Storage

- TLS implemented by __thread
  - alg.14-1-tls-thread.c (1)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/syscall.h>
#include <pthread.h>
#define gettid() syscall(__NR_gettid)

__thread int tlsvar = 0; /* tlsvar for each thread; interpreted by language compiler,
    a language level solution to thread local storage */

static void* thread_worker(void* arg)
{
    char *param = (char *)arg;
    int randomcount;

    for (int i = 0; i < 5; ++i) {
        randomcount = rand() % 100000;
        for (int k = 0; k < randomcount; k++) ;
        printf("%s%ld, tlsvar = %d\n", param, gettid(), tlsvar);
        tlsvar++; /* each thread has its local tlsvar */
    }

    pthread_exit(0);

}
```

Thread Local Storage

- TLS implemented by __thread
  - alg.14-1-tls-thread.c (2)

```
int main(void)
{
    pthread_t tid1, tid2;
    char para1[] = "                    ";
    char para2[] = "                                        ";
    int randomcount;

    pthread_create(&tid1, NULL, &thread_worker, para1);
    pthread_create(&tid2, NULL, &thread_worker, para2);

    printf("parent               tid1                 tid2\n");
    printf("================     ================     ================\n");

    for (int i = 0; i < 5; ++i) {
        randomcount = rand() % 100000;
        for (int k = 0; k < randomcount; k++) ;
        printf("%ld, tlsvar = %d\n", gettid(), tlsvar);
        tlsvar++; /* main- thread has its local tlsvar */
    }

    sleep(1);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    return 0;
}
```

Thread Local Storage

- TLS implemented by __thread
  - alg.14-1-tls-thread.c (2)

```
isscgy@ubuntu:/mnt/os-2020$ gcc alg.14-1-tls-thread.c -pthread
isscgy@ubuntu:/mnt/os-2020$ ./a.out
parent                    tid1                      tid2
================          ================          ================
20914, tlsvar = 0
20914, tlsvar = 1
                          20915, tlsvar = 0
                                                    20916, tlsvar = 0
20914, tlsvar = 2
                          20915, tlsvar = 1
                          20915, tlsvar = 2
                                                    20916, tlsvar = 1
                                                    20916, tlsvar = 2
                          20915, tlsvar = 3
20914, tlsvar = 3
                          20915, tlsvar = 4
20914, tlsvar = 4
                                                    20916, tlsvar = 3
                                                    20916, tlsvar = 4
isscgy@ubuntu:/mnt/os-2020$
```

Thread Local Storage

- TLS implemented by pthread_key_create

```
typedef unsigned int pthread_key_t
```

  - Keys for thread-specific data
  - Check the *upper bound* of Thread-specific Data (TSD) in sysdeps/x86/bits/pthreadtypes.h

```
PTHREAD_KEYS_MAX = 1024
```

  - A process can at most create 1024 Keys

```
int pthread_key_create(pthread_key_t *key, void
(*destructor)(void*)); /* tls_key is an identifier rather
than a data structure */
int pthread_setspecific(pthread_key_t key, const void
*value);
void *pthread_getspecific(pthread_key_t key);
int pthread_key_delete(pthread_key_t key); */
```

Thread Local Storage

- TLS implemented by pthread_key_create
  - alg.14-2-tls-pthread-key-1.c (1)

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/syscall.h>
#include <unistd.h>
#include <pthread.h>
#define gettid() syscall(__NR_gettid)
```

```
/* gcc -pthread */
static pthread_key_t log_key;
/* each thread associates global log_key with one local variable of the thread */
/* the associated local variable behaves like a global variable by use of log_key */
void write_log(const char *msg)
{
    FILE *fp_log;
    fp_log = (FILE *)pthread_getspecific(log_key); /* fp_log obtained from log_key */
    fprintf(fp_log, "writing msg: %s\n", msg);
    printf("log_key = %d, tid = %ld, address of fp_log %p\n", log_key, gettid(),
fp_log);
}

static void *thread_worker(void *args)
{
    static int thcnt = 0;
    char fname[64], msg[64];
    FILE *fp_log; /* a local variable */

    sprintf(fname, "log/thread-%d.log", ++thcnt);  /* directory ./log must exist */
    fp_log = fopen(fname, "w");
    if(!fp_log) {
        printf("%s\n", fname);
        perror("fopen()");
        return NULL;
    }
```

Thread Local Storage

- TLS implemented by pthread_key_create
  - alg.14-2-tls-pthread-key-1.c (2)

```
    pthread_setspecific(log_key, fp_log); /* fp_log is associated with log_key */

    sprintf(msg, "Here is %s\n", fname);
    write_log(msg);
}

void close_log_file(void* log_file) /* the destructor */
{
    fclose((FILE*)log_file);
}
```

Thread Local Storage

- TLS implemented by pthread_key_create
  - alg.14-2-tls-pthread-key-1.c (3)

```
int main(void)
{
    const int n = 5;
    pthread_t tids[n];
    pthread_key_create(&log_key, &close_log_file);
// or:    pthread_key_create(&log_key, NULL); /* NULL for default destructor */

    printf("======tids and TLS variable addresses ======\n");
    for(int i = 0; i < n; i++) {
        pthread_create(&tids[i], NULL, &thread_worker, NULL);
    }

    for(int i = 0; i < n; i++) {
        pthread_join(tids[i], NULL);
    }

    pthread_key_delete(log_key); /* delete the key */

    printf("\ncommand: lsof +d ./log\n");
    system("lsof +d ./log"); /* list all open instance of directory ./log */
    printf("\ncommand: cat ./log/thread-1.log ./log/thread-5.log\n");
    system("cat ./log/thread-1.log ./log/thread-5.log");

    return 0;
}
```

Thread Local Storage

- TLS implemented by pthread_key_create
  - alg.14-2-tls-pthread-key-1.c (3)

```
isscgy@ubuntu:/mnt/os-2020$ gcc alg.14-2-tls-pthread-key-1.c -pthread
isscgy@ubuntu:/mnt/os-2020$ ./a.out
======tids and TLS variable addresses ======
log_key = 0, tid = 47249, address of fp_log 0x7f2a20000b20
log_key = 0, tid = 47250, address of fp_log 0x7f2a18000b20
log_key = 0, tid = 47251, address of fp_log 0x7f2a1c000b20
log_key = 0, tid = 47253, address of fp_log 0x7f2a14000b20
log_key = 0, tid = 47252, address of fp_log 0x7f2a10000b20

command: lsof +d ./log

command: cat ./log/thread-1.log ./log/thread-5.log
writing msg: Here is log/thread-1.log

writing msg: Here is log/thread-5.log

isscgy@ubuntu:/mnt/os-2020$
```

Different fp_log with the same log_key

```
    return 0;
}
```

Thread Local Storage

- TLS implemented by pthread_key_create
  - alg.14-2-tls-pthread-key-1.c (3)

```
isscgy@ubuntu:/mnt/os-2020$ gcc alg.14-2-tls-pthread-key-1.c -pthread
isscgy@ubuntu:/mnt/os-2020$ ./a.out
======tids and TLS variable addresses ======
log_key = 0, tid = 47249, address of fp_log 0x7f2a20000b20
log_key = 0, tid = 47250, address of fp_log 0x7f2a18000b20
log_key = 0, tid = 47251, address of fp_log 0x7f2a1c000b20
log_key = 0, tid = 47253, address of fp_log 0x7f2a14000b20
log_key = 0, tid = 47252, address of fp_log 0x7f2a10000b20

command: lsof +d ./log

command: cat ./log/thread-1.log ./log/thread-5.log
writing msg: Here is log/thread-1.log

writing msg: Here is log/thread-5.log

isscgy@ubuntu:/mnt/os-2020$
```

All log files closed by the destructor “void close_log_file()”

```
    return 0;
}
```

Thread Local Storage

- TLS implemented by pthread_key_create
  - alg.14-3-tls-pthread-key-2.c (1): binding data structures

```
/* gcc -pthread */
#include <stdio.h>
#include <stdlib.h>
#include <sys/syscall.h>
#include <unistd.h>
#include <malloc.h>
#include <pthread.h>
#define gettid() syscall(__NR_gettid)

static pthread_key_t tls_key; /* static global */

void print_msg1(void);
void print_msg2(void);
static void *thread_func1(void *);
static void *thread_func2(void *);

/* msg1 and mag2 have different structures */
struct msg_struct1 {
    char stuno[9];
    char stuname[20];
};
struct msg_struct2 {
    int stuno;
    char nationality[20];
    char stuname[20];
};
```

Thread Local Storage

- TLS implemented by pthread_key_create
  - alg.14-3-tls-pthread-key-2.c (2): binding data structures

```
int main(void)
{
    pthread_t ptid1, ptid2;

    pthread_key_create(&tls_key, NULL);

    printf("      msg1 -->>        stuno     stuname       msg2 -->>         stuno
stuname  nationaluty\n");
    printf("=================== ========= ======== =================== =========
======== ============\n");

    pthread_create(&ptid1, NULL, &thread_func1, NULL);
    pthread_create(&ptid2, NULL, &thread_func2, NULL);

    pthread_join(ptid1, NULL);
    pthread_join(ptid2, NULL);

    pthread_key_delete(tls_key);
    return EXIT_SUCCESS;
}
```

Thread Local Storage

- TLS implemented by pthread_key_create
  - alg.14-3-tls-pthread-key-2.c (3): binding data structures

```
static void *thread_func1(void *args)
{
    struct msg_struct1 ptr[5]; /* local variable in thread stacke */
    printf("thread_func1: tid = %ld   ptr = %p\n", gettid(), ptr);
    pthread_setspecific(tls_key, ptr); /* binding ptr to the tls_key */
    sprintf(ptr[0].stuno, "18000001"); sprintf(ptr[0].stuname, "Alex");
    sprintf(ptr[4].stuno, "18000005"); sprintf(ptr[4].stuname, "Michael");
    print_msg1();

    pthread_exit(0);
}

void print_msg1(void)
{
    int randomcount;
    struct msg_struct1 *ptr = (struct msg_struct1 *)pthread_getspecific(tls_key);
    printf("print_msg1:   tid = %ld   ptr = %p\n", gettid(), ptr);
    for (int i = 1; i < 6; i++) {
        randomcount = rand() % 10000;
        for (int k =0; k < randomcount; k++) ;
        printf("tid = %ld  i = %2d   %s  %*.*s\n", gettid(), i, ptr->stuno, 8, 8, ptr-
>stuname);
        ptr++;
    }
    return;
}
```

Thread Local Storage

- TLS implemented by pthread_key_create
  - alg.14-3-tls-pthread-key-2.c (4): binding data structures

```
static void *thread_func2(void *args)
{   struct msg_struct2 *ptr;
    ptr = (struct msg_struct2 *)malloc(5*sizeof(struct msg_struct2)); /* storage in
process heap */
    printf("thread_func2: tid = %ld   ptr = %p\n", gettid(), ptr);
    pthread_setspecific(tls_key, ptr);
    ptr->stuno = 19000001; sprintf(ptr->stuname, "Bob");
    sprintf(ptr->nationality, "United Kingdom");
    (ptr+2)->stuno = 19000003; sprintf((ptr+2)->stuname, "John");
    sprintf((ptr+2)->nationality, "United States");
    print_msg2();
    free(ptr); ptr = NULL; pthread_exit(0);
}
void print_msg2(void)
{   int randomcount;
    struct msg_struct2* ptr = (struct msg_struct2 *)pthread_getspecific(tls_key);
    printf("print_msg2:   tid = %ld   ptr = %p\n", gettid(), ptr);
    for (int i = 1; i < 6; i++) {
        randomcount = rand() % 10000;
        for (int k =0; k < randomcount; k++) ;
        printf("                                          tid = %ld  i = %2d   %d
%*.*s   %s\n", gettid(), i, ptr->stuno, 8, 8, ptr->stuname, ptr->nationality);
        ptr++;
    }
    return;
}
```

Thread Local Storage

- TLS implemented by pthread_key_create
  - alg.14-3-tls-pthread-key-2.c (4): binding data structures

```
isscgy@ubuntu:/mnt/os-2020$ gcc alg.14-3-tls-pthread-key-2.c -pthread
isscgy@ubuntu:/mnt/os-2020$ ./a.out
      msg1 -->>        stuno     stuname       msg2 -->>         stuno     stuname   nationaluty
===================  =========  ========  ===================  =========  ========  ============
thread_func1: tid = 47333   ptr = 0x7fa8b549be50
print_msg1:   tid = 47333   ptr = 0x7fa8b549be50
tid = 47333  i =  1   18000001      Alex
tid = 47333  i =  2
tid = 47333  i =  3
tid = 47333  i =  4
tid = 47333  i =  5   18000005   Michael
thread_func2: tid = 47334   ptr = 0x7fa8b0000b20
print_msg2:   tid = 47334   ptr = 0x7fa8b0000b20
                                          tid = 47334  i =  1   19000001       Bob   United Kingdom
                                          tid = 47334  i =  2   0
                                          tid = 47334  i =  3   19000003      John   United States
                                          tid = 47334  i =  4   0
                                          tid = 47334  i =  5   0
isscgy@ubuntu:/mnt/os-2020$ 
```

```
        for (int k =0; k < randomcount; k++) ;
        printf("                                               tid = %ld  i = %2d   %d
%*.*s   %s\n", gettid(), i, ptr->stuno, 8, 8, ptr->stuname, ptr->nationality);
        ptr++;
    }
    return;
}
```

Thread Local Storage

- TLS implemented by pthread_key_create
  - alg.14-4-tls-pthread-key-3.c (1): functions of threads

```
/* gcc -pthread */
#include <stdio.h>
#include <stdlib.h>
#include <sys/syscall.h>
#include <unistd.h>
#include <pthread.h>
#define gettid() syscall(__NR_gettid)

static pthread_key_t tls_key; /* static global */

void print_msg(void);
static void *thread_func1(void *);
static void *thread_func2(void *);

struct msg_struct {
    char pos[80];
    char stuno[9];
    char stuname[20];
};
```

Thread Local Storage

- TLS implemented by pthread_key_create
  - alg.14-4-tls-pthread-key-3.c (2): functions of threads

```
int main(void)
{
    pthread_t ptid1, ptid2;

    pthread_key_create(&tls_key, NULL);

    printf("       msg1 -->>        stuno     stuname       msg2 -->>         stuno
stuname\n");
    printf(" ===================  =========  ========  ===================  =========
========\n");

    pthread_create(&ptid1, NULL, &thread_func1, NULL);
    pthread_create(&ptid2, NULL, &thread_func2, NULL);

    pthread_join(ptid1, NULL);
    pthread_join(ptid2, NULL);

    pthread_key_delete(tls_key);
    return EXIT_SUCCESS;
}
```

Thread Local Storage

- TLS implemented by pthread_key_create
  - alg.14-4-tls-pthread-key-3.c (3): functions of threads

```
static void *thread_func1(void *args)
{
    struct msg_struct ptr[5]; /* in thread stack */
    printf("thread_func1: tid = %ld   ptr = %p\n", gettid(), ptr);

    pthread_setspecific(tls_key, ptr); /* binding its tls_key to address of ptr */
    for (int i =0; i<5; i++) {
        sprintf(ptr[i].pos, " ");
        sprintf(ptr[i].stuno, "        ");
        sprintf(ptr[i].stuname, "                   ");
    }
    sprintf(ptr[0].stuno, "18000001");
    sprintf(ptr[0].stuname, "Alex");
    sprintf(ptr[4].stuno, "18000005");
    sprintf(ptr[4].stuname, "Michael");
    print_msg(); /* thread_func1 and thread_fun2 call the same print_msg() */
    pthread_exit(0);
}
```

Thread Local Storage

- TLS implemented by pthread_key_create
  - alg.14-4-tls-pthread-key-3.c (4): functions of threads

```
static void *thread_func2(void *args)
{
    struct msg_struct ptr[5]; /* in thread stack */
    printf("thread_func2: tid = %ld   ptr = %p\n", gettid(), ptr);

    pthread_setspecific(tls_key, ptr); /* binding its tls_key to address of ptr */

    for (int i = 0;  i < 5; i++) {
        sprintf(ptr[i].pos, "                                          ");
        sprintf(ptr[i].stuno, "        ");
        sprintf(ptr[i].stuname, "                   ");
    }
    sprintf(ptr[0].stuno, "19000001");
    sprintf(ptr[0].stuname, "Bob");
    sprintf(ptr[2].stuno, "19000003");
    sprintf(ptr[2].stuname, "John");
    print_msg(); /* thread_func1 and thread_fun2 call the same print_msg() */
    pthread_exit(0);
}
```

Thread Local Storage

- TLS implemented by pthread_key_create
  - alg.14-4-tls-pthread-key-3.c (5): functions of threads

```
void print_msg(void)
{
    int randomcount;

    struct msg_struct* ptr = (struct msg_struct *)pthread_getspecific(tls_key);
        /* ptr decided by call thread */
    printf("print_msg:    tid = %ld   ptr = %p\n", gettid(), ptr);

    for (int i = 1; i < 6; i++) {
        randomcount = rand() % 10000;
        for (int k =0; k < randomcount; k++) ;
        printf("%stid = %ld  i = %2d   %s  %*.*s\n", ptr->pos, gettid(), i, ptr->stuno,
8, 8, ptr->stuname);
        ptr++;
    }

    return;
}
```

Thread Local Storage

- TLS implemented by pthread_key_create
  - alg.14-4-tls-pthread-key-3.c (5): functions of threads

```
isscgy@ubuntu:/mnt/os-2020$ gcc alg.14-4-tls-pthread-key-3.c -pthread
isscgy@ubuntu:/mnt/os-2020$ ./a.out
      msg1 -->>          stuno     stuname        msg2 -->>          stuno     stuname
 ===================  =========  ========  ===================  =========  ========
thread_func1: tid = 47508   ptr = 0x7fd4fe4b3cc0
print_msg:    tid = 47508   ptr = 0x7fd4fe4b3cc0
thread_func2: tid = 47509   ptr = 0x7fd4fdcb2cc0
print_msg:    tid = 47509   ptr = 0x7fd4fdcb2cc0
                                           tid = 47509  i =  1   19000001       Bob
 tid = 47508  i =  1   18000001      Alex
                                           tid =
 tid = 47508  i =  2
                                           tid =                                 nn
                                           tid = 47509  i =  4
 tid = 47508  i =  3
                                           tid = 47509  i =  5
 tid = 47508  i =  4
 tid = 47508  i =  5   18000005   Michael
isscgy@ubuntu:/mnt/os-2020$
```

func1 and func2 call print_msg() which has two copies of ptr for func1 and func2 respectively

Thread Local Storage

- TLS implemented by pthread_key_create
  - alg.14-5-tls-pthread-key-4.c (1): set tls_key in subfunctions

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/syscall.h>
#include <unistd.h>
#include <malloc.h>
#include <pthread.h>
#define gettid() syscall(__NR_gettid)
static pthread_key_t tls_key; /* static global */
static void *thread_func(void *);
void thread_data1(void);
void thread_data2(void);

struct msg_struct {
    char stuno[9];
    char stuname[20];
};

int main(void)
{
    pthread_t ptid;
    pthread_key_create(&tls_key, NULL);
    pthread_create(&ptid, NULL, &thread_func, NULL);
    pthread_join(ptid, NULL);
    pthread_key_delete(tls_key);
    return EXIT_SUCCESS;
}
```

Thread Local Storage

- TLS implemented by pthread_key_create
  - alg.14-5-tls-pthread-key-4.c (2): set tls_key in subfunctions

```
static void *thread_func(void *args)
{
    struct msg_struct *ptr;

    thread_data1();
    ptr = (struct msg_struct *)pthread_getspecific(tls_key);
        /* get ptr from thread_data1() */
    perror("pthread_getspecific()");
    printf("ptr from thread_data1() in thread_func(): %p\n", ptr);
    for (int i = 1; i < 6; i++) {
        printf("tid = %ld  i = %2d   %s  %*.*s\n", gettid(), i, (ptr+i-1)->stuno, 8, 8, 
(ptr+i-1)->stuname);
    }
    thread_data2();
    ptr = (struct msg_struct *)pthread_getspecific(tls_key);
       /* get ptr from thread_data2() */
    perror("pthread_getspecific()");
    printf("ptr from thread_data2() in thread_func(): %p\n", ptr);
    for (int i = 1; i < 6; i++) {
        printf("tid = %ld  i = %2d   %s  %*.*s\n", gettid(), i, (ptr+i-1)->stuno, 8, 8, 
(ptr+i-1)->stuname);
    }
    free(ptr);
    ptr = NULL;
    pthread_exit(0);
}
```

Thread Local Storage

- TLS implemented by pthread_key_create
  - alg.14-5-tls-pthread-key-4.c (3): set tls_key in subfunctions

```
void thread_data1(void)
{
    struct msg_struct ptr[5]; /* in thread stack */
    pthread_setspecific(tls_key, ptr); /* binding the tls_key to address of ptr */
    printf("ptr in thread_data1(): %p\n", ptr);

    for (int i = 0; i < 5; i++) {
        sprintf(ptr[i].stuno, "        ");
        sprintf(ptr[i].stuname, "                  ");
    }
    sprintf(ptr[0].stuno, "19000001");
    sprintf(ptr[0].stuname, "Bob");
    sprintf(ptr[2].stuno, "19000003");
    sprintf(ptr[2].stuname, "John");

    return;
    /* thread stack space is deallocated when thread_data1() returns and data lost */
}
```

Thread Local Storage

- TLS implemented by pthread_key_create
  - alg.14-5-tls-pthread-key-4.c (4): set tls_key in subfunctions

```
void thread_data2(void)
{
    struct msg_struct *ptr;
    ptr = (struct msg_struct *)malloc(5*sizeof(struct msg_struct));
        /* in process heap */
    pthread_setspecific(tls_key, ptr); /* binding the tls_key to address of ptr */
    printf("ptr in thread_data2(): %p\n", ptr);

    for (int i = 0; i < 5; i++) {
        sprintf(ptr[i].stuno, "        ");
        sprintf(ptr[i].stuname, "                   ");
    }
    sprintf(ptr->stuno, "19000001");
    sprintf(ptr->stuname, "Bob");
    sprintf((ptr+2)->stuno, "19000003");
    sprintf((ptr+2)->stuname, "John");

    return;
    /* the heap space is kept effective if ptr is not freed */
    /* if free(ptr) before return, the space is reallocated and data lost */
    /* need to free the space in thread_func or there is a memory leak */
}
```

Thread Local Storage

- TLS implemented by pthread_key_create
  - alg.14-5-tls-pthread-key-4.c (4): set tls_key in subfunctions

```
isscgy@ubuntu:/mnt/os-2020$ gcc alg.14-5-tls-pthread-key-4.c -pthread
isscgy@ubuntu:/mnt/os-2020$ ./a.out
ptr in thread_data1(): 0x7ffb277d9e10
pthread_getspecific(): Success
ptr from thread_data1() in thread_func(): 0x7ffb277d9e10
tid = 47607  i =  1
tid = 47607  i =  2
tid = 47607  i =  3   �'�       '�
tid = 47607  i =  4
tid = 47607  i =  5
ptr in thread_data2(): 0x7ffb20001570
pthread_getspecific(): Success
ptr from thread_data2() in thread_func(): 0x7ffb20001570
tid = 47607  i =  1   19000001       Bob
tid = 47607  i =  2
tid = 47607  i =  3   19000003      John
tid = 47607  i =  4
tid = 47607  i =  5
isscgy@ubuntu:/mnt/os-2020$
```

In both cases tls_key can keep working

Thread Local Storage

- TLS implemented by pthread_key_create
  - alg.14-5-tls-pthread-key-4.c (4): set tls_key in subfunctions

```
isscgy@ubuntu:/mnt/os-2020$ gcc alg.14-5-tls-pthread-key-4.c -pthread
isscgy@ubuntu:/mnt/os-2020$ ./a.out
ptr in thread_data1(): 0x7ffb277d9e10
pthread_getspecific(): Success
ptr from thread_data1() in thread_func(): 0x7ffb277d9e10
tid = 47607  i =  1
tid = 47607  i =  2
tid = 47607  i =  3   �'�         '�
tid = 47607  i =  4
tid = 47607  i =  5
ptr in thread_data2(): 0x7ffb20001570
pthread_getspecific(): Success
ptr from thread_data2() in thread_func(): 0x7ffb20001570
tid = 47607  i =  1   19000001       Bob
tid = 47607  i =  2
tid = 47607  i =  3   19000003      John
tid = 47607  i =  4
tid = 47607  i =  5
isscgy@ubuntu:/mnt/os-2020$
```

Data lost

Data effective

Scheduler Activations

- Both Many-to-Many and Two-level threading models require communication to maintain the appropriate number of kernel threads allocated to the application for better performance.
- Typically use an intermediate data structure between user and kernel threads – lightweight process (LWP):
  - Appears to be a virtual processor on which process can schedule user thread to run.
  - Each LWP attached to kernel thread.
  - How many LWPs to create?
- Scheduler activations provide *upcalls* – a communication mechanism from the kernel to the upcall handler in the thread library.
  - the kernel informs an application about certain events by upcall.
  - This communication allows an application to maintain the correct number kernel threads.

Linux clone()

- Linux provides fork() and vfork() system calls with the traditional functionality of duplicating a process. Linux also provides the ability to create threads using the clone() system call.
  - In fact, Linux uses the term *task*—rather than *process* or *thread*—when referring to a flow of control within a program. It does not distinguish between processes and threads.
  - clone() with a set of flags allows a child task to share some resources of the parent task. The flags determine how much sharing is to take place between the parent and child tasks.
  - If none of these flags is set when clone() is invoked, no sharing takes place, similar to that provided by the fork() system call.

flag	meaning
CLONE_FS	File-system information is shared
CLONE_VM	The same memory space is shared
CLONE_SIGHAND	Signal handlers are shared
CLONE_FILES	The set of open files is shared

Linux clone()

- The data structure task_struct
  - A Linux kernel data structure struct task_struct exists for each task in the system. This data structure, instead of storing data for the task, contains pointers to other data structures where these data are stored.
    - E.g., data structures that represent the list of open files, signal-handling information, and virtual memory.
    - Check the file sched.h in your system for task_struct. e.g.,
      /usr/src/linux-headers-5.3.0-53/include/linux# vim sched.h
  - When fork() is invoked, a new task is created, along with a copy of all the associated data structures of the parent process.
  - When the clone() system call is made, a new task is also created. However, rather than copying all data structures, pointers of the task_struct of the new task point to the genuine, or the duplicated, data structures of the parent, depending on the set of flags passed to clone().

Linux clone()

- Prototype of clone()

```
#define _GNU_SOURCE
#include <sched.h>

int clone(int (*fn)(void *), void *child_stack, int flags, void
*arg, ... /* pid_t *ptid, struct user_desc *tls, pid_t *ctid */
);
```

- It is actually a library function layered on top of the underlying clone() system call, hereinafter referred to as sys_clone system call.
- The main use of clone() is to implement threads: multiple threads of control in a program that run concurrently in a shared memory space.
- When the child process is created with clone(), it executes the function *fn*(*arg*).
- When the *fn*(*arg*) function application returns, the child process/task terminates. The integer returned by *fn* is the exit code for the child process. The child process may also terminate explicitly by calling exit or after receiving a fatal signal.

Linux clone()

- Prototype of clone()

```
#define _GNU_SOURCE
#include <sched.h>

int clone(int (*fn)(void *), void *child_stack, int flags, void
*arg, ... /* pid_t *ptid, struct user_desc *tls, pid_t *ctid */
);
```

- The *child_stack* argument specifies the location of the stack used by the child process.
  - Since the child and calling process may share memory, it is not possible for the child process to execute in the same stack as the calling process.
  - The calling process must therefore set up memory space for the child stack and pass a pointer to this space to clone().
  - Stacks *grow downward* on all processors that run Linux (except the HP PA processors), so *child_stack* usually points to the *topmost address* of the memory space set up for the child stack.

Linux clone()

- Prototype of clone()

```
#define _GNU_SOURCE
#include <sched.h>

int clone(int (*fn)(void *), void *child_stack, int flags, void
*arg, ... /* pid_t *ptid, struct user_desc *tls, pid_t *ctid */
);
```

- The low byte of *flags* contains the number of the termination signal sent to the parent when the child dies.
  - If this signal is specified as anything other than SIGCHLD, then the parent process must specify the __WALL or __WCLONE options when waiting for the child with wait().
  - If no signal is specified, then the parent process is not signaled when the child terminates.
- *flags* may also be bitwise-or‘ed (按位或运算) with zero or more of the following constants specified from the next slide, in order to specify what is shared between the calling task and the child task.

Linux clone()

- Check the Linux manual:

```
#man 2 clone
https://linux.die.net/man/2/clone
http://man7.org/linux/man-pages/man2/clone.2.html
https://www.kernel.org/doc/man-pages/
```

- The Linux *man-pages* project
  1. User commands; man-pages includes a very few Section 1 pages that document programs supplied by the GNU C library.
  2. System calls documents the system calls provided by the Linux kernel.
  3. Library functions documents the functions provided by the standard C library.
  4. Devices documents details of various devices, most of which reside in /dev.
  5. Files describes various file formats, and includes proc(5), which documents the /proc file system.
  7. Overviews, conventions, and miscellaneous.
  8. Superuser and system administration commands; man-pages includes a very few Section 8 pages that document programs supplied by the GNU C library.

Linux clone()

- Constants with flags in clone()
  - CLONE_PARENT
    - If CLONE_PARENT is set, then the parent of the new child (as returned by getppid(2)) will be the same as that of the calling process (i.e., a newborn sibling of the calling).
    - If CLONE_PARENT is not set, then (as with fork(2)) the child's parent is the calling process.
    - Note that it is the parent process, as returned by getppid(2), which is signaled (SIGCHLD) when the child terminates. Setting CLONE_PARENT will make the parent of the calling process, rather than the calling process itself, being signaled.

Linux clone()

- Constants with flags in clone()
  - CLONE_NEWPID
    - If CLONE_NEWPID is set, then create the process in a new PID namespace. If this flag is not set, then the process is created in the same PID namespace as the calling process. This flag is intended for the implementation of containers.
      - A PID namespace provides an isolated environment for PIDs: PIDs in a new namespace start at 1, somewhat like a standalone system, and calls to fork(2), vfork(2), or clone() will produce processes with PIDs that are unique within the namespace.
    - The first process created in a new namespace (i.e., the process created using the CLONE_NEWPID flag) has the PID 1, and is the "init" process for the namespace. Children that are orphaned within the namespace will be reparented to this process rather than init(8). Unlike the traditional init process, the "init" process of a PID namespace can terminate, and if it does, all of the processes in the namespace are terminated.

Linux clone()

- Constants with flags in clone()
  - CLONE_NEWPID (2)
    - PID namespaces form a hierarchy. When a new PID namespace is created, the processes in that namespace are visible in the PID namespace of the process that created the new namespace; analogously, if the parent PID namespace is itself the child of another PID namespace, then processes in the child and parent PID namespaces will both be visible in the grandparent PID namespace. Conversely, the processes in the "child" PID namespace do not see the processes in the parent namespace. The existence of a namespace hierarchy means that each process may now have multiple PIDs: one for each namespace in which it is visible; each of these PIDs is unique within the corresponding namespace. (A call to getpid(2) always returns the PID associated with the namespace in which the process lives.)

Linux clone()

- Constants with flags in clone()
  - CLONE_NEWPID (3)
    - After creating the new namespace, it is useful for the child to change its root directory and mount a new procfs instance at */proc* so that tools such as ps(1) work correctly. (If CLONE_NEWNS is also included in *flags*, then it isn't necessary to change the root directory: a new procfs instance can be mounted directly over */proc*.)
    - Use of this flag requires: a kernel configured with the CONFIG_PID_NS option and that the process be privileged (CAP_SYS_ADMIN). This flag can't be specified in conjunction with CLONE_THREAD.

Linux clone()

- Constants with flags in clone()
  - CLONE_FS
    - If CLONE_FS is set, the caller and the child process share the same file system information.
      - This includes the root of the file system, the current working directory, and the umask.
      - Any call to chroot(2), chdir(2), or unmask(2) performed by the calling process or the child process also affects the other process.
    - If CLONE_FS is not set, the child process works on a copy of the file system information of the calling process at the time of the clone() call. Calls to chroot(2), chdir(2), unmask(2) performed later by one of the processes do not affect the other process.

Linux clone()

- Constants with flags in clone()
  - CLONE_FILES
    - If CLONE_FILES is set, the calling process and the child process share the same file descriptor table.
      - Any file descriptor created by the calling process or by the child process is also valid in the other process.
      - Similarly, if one of the processes closes a file descriptor, or changes its associated flags (using the fcntl(2) F_SETFD operation), the other process is also affected.
    - If CLONE_FILES is not set, the child process inherits a copy of all file descriptors opened in the calling process at the time of clone(). (The duplicated file descriptors in the child refer to the same open file descriptions as the corresponding file descriptors in the calling process.) Subsequent operations that open or close file descriptors, or change file descriptor flags, performed by either the calling process or the child process do not affect the other process.

Linux clone()

- Constants with flags in clone()
  - CLONE_NEWNS
    - Every process lives in a mount namespace which is the set of mounts describing the file hierarchy as seen by that process.
    - After a fork(2) or clone() where the CLONE_NEWNS flag is not set, the child lives in the same mount namespace as the parent.
      - The system calls mount(2) and umount(2) change the mount namespace of the calling process, and hence affect all processes that live in the same namespace, but do not affect processes in a different mount namespace.
    - After a clone() where the CLONE_NEWNS flag is set, the cloned child is started in a new mount namespace, initialized with a copy of the namespace of the parent.
    - Only a privileged process (one having the CAP_SYS_ADMIN capability) may specify the CLONE_NEWNS flag. It is not permitted to specify both CLONE_NEWNS and CLONE_FS in the same clone() call.

Linux clone()

- Constants with flags in clone()
  - CLONE_SIGHAND
    - If CLONE_SIGHAND is set, the calling process and the child process share the same table of signal handlers. If the calling process or child process calls sigaction(2) to change the behavior associated with a signal, the behavior is changed in the other process as well. However, the calling process and child processes still have distinct signal masks and sets of pending signals. So, one of them may block or unblock some signals using sigpromask(2) without affecting the other process.
    - If CLONE_SIGHAND is not set, the child process inherits a copy of the signal handlers of the calling process at the time clone() is called. Calls to sigaction(2) performed later by one of the processes have no effect on the other process.
    - Since Linux 2.6.0-test6, *flags* must also include CLONE_VM if CLONE_SIGHAND is specified.

- **Linux clone()**
  - Constants with flags in clone()
    - CLONE_PTRACE
      - If CLONE_PTRACE is specified, and the calling process is being traced, then trace the child also.
    - CLONE_UNPTRACE
      - If CLONE_UNTRACED is specified, then a tracing process cannot force CLONE_PTRACE on this child process.

Linux clone()

- Constants with flags in clone()
  - CLONE_VFORK
    - If CLONE_VFORK is set, the execution of the calling process is *suspended* until the child releases its virtual memory resources via a call to execve(2) or _exit(2) (as with vfork(2)).
    - If CLONE_VFORK is not set then both the calling process and the child are schedulable after the call, and an application should not rely on execution occurring in any particular order.

Linux clone()

- Constants with flags in clone()
  - CLONE_VM
    - If CLONE_VM is set, the calling process and the child process run in the same memory space. In particular, memory writes performed by the calling process or by the child process are also visible in the other process. Moreover, any memory mapping or unmapping performed with mmap(2) or munmap(2) by the child or calling process also affects the other process.
    - If CLONE_VM is not set, the child process runs in a separate copy of the memory space of the calling process at the time of clone(). Memory writes or file mappings/unmappings performed by one of the processes do not affect the other, as with fork(2).

Linux clone()

- Constants with flags in clone()
  - CLONE_IO
    - If CLONE_IO is set, then the new process shares an I/O context with the calling process.
      - The I/O context is the I/O scope of the disk scheduler (i.e, what the I/O scheduler uses to model scheduling of a process's I/O). If processes share the same I/O context, they are treated as one by the I/O scheduler. As a consequence, they get to share disk time. For some I/O schedulers, if two processes share an I/O context, they will be allowed to interleave their disk access. If several threads are doing I/O on behalf of the same process (aio_read(3), for instance), they should employ CLONE_IO to get better I/O performance.
      - If the kernel is not configured with the CONFIG_BLOCK option, this flag is a no-op.
    - If CLONE_IO is not set, then (as with fork(2)) the new process has its own I/O context.

Linux clone()

- Constants with flags in clone()
  - CLONE_NEWUTS
    - If CLONE_NEWUTS is set, then create the process in a new UTS namespace, whose identifiers are initialized by duplicating the identifiers from the UTS namespace of the calling process.
    - If this flag is not set, then (as with fork(2)) the process is created in the same UTS namespace as the calling process. This flag is intended for the implementation of containers.
    - A UTS namespace is the set of identifiers returned by uname(2); among these, the domain name and the host name can be modified by setdomainname(2) and sethostname(2), respectively. Changes made to the identifiers in a UTS namespace are visible to all other processes in the same namespace, but are not visible to processes in other UTS namespaces. (see uts_namespaces(7))
    - Use of this flag requires: a kernel configured with the CONFIG_UTS_NS option and that the process be privileged (CAP_SYS_ADMIN).

Linux clone()

- Constants with flags in clone()
  - CLONE_NEWUSER
    - If CLONE_NEWUSER is set, then create the process in a new user namespace. If this flag is not set, then (as with fork(2)) the process is created in the same user namespace as the call- ing process.
    - For further information on user namespaces, see namespaces(7) and user_namespaces(7).
    - Before Linux 3.8, use of CLONE_NEWUSER required that the caller have three capabilities: CAP_SYS_ADMIN, CAP_SETUID, and CAP_SETGID. Starting with Linux 3.8, no privileges are needed to create a user namespace.
    - This flag can't be specified in conjunction with CLONE_THREAD or CLONE_PARENT. For security reasons, CLONE_NEWUSER cannot be specified in conjunction with CLONE_FS.

Linux clone()

- Constants with flags in clone()
  - CLONE_NEWIPC
    - If CLONE_NEWIPC is set, then create the process in a new IPC namespace.
    - If CLONE_NEWIPC is not set, then (as with fork(2)), the process is created in the same IPC namespace as the calling process.
    - This flag is intended for the implementation of containers. An IPC namespace provides an isolated view of System V IPC objects and (since Linux 2.6.30) POSIX message queues. The common characteristic of these IPC mechanisms is that IPC objects are identified by mechanisms other than filesystem pathnames.
    - Objects created in an IPC namespace are visible to all other processes that are members of that namespace, but are not visible to processes in other IPC namespaces.

Linux clone()

- Constants with flags in clone()
  - CLONE_NEWIPC (2)
    - When an IPC namespace is destroyed (i.e., when the last process that is a member of the namespace terminates), all IPC objects in the namespace are automatically destroyed.
    - Use of this flag requires: a kernel configured with the CONFIG_SYSVIPC and CONFIG_IPC_NS options and that the process be privileged (CAP_SYS_ADMIN). This flag can't be specified in conjunction with CLONE_SYSVSEM.

Linux clone()

- Constants with flags in clone()
  - CLONE_NEWNET
    - (The implementation of this flag was only completed by about kernel version 2.6.29.)
    - If CLONE_NEWNET is set, then create the process in a new network namespace.
    - If CLONE_NEWNET is not set, then (as with fork(2)), the process is created in the same network namespace as the calling process.
    - This flag is intended for the implementation of containers. A network namespace provides an isolated view of the networking stack (network device interfaces, IPv4 and IPv6 protocol stacks, IP routing tables, firewall rules, the */proc/net* and */sys/class/net* directory trees, sockets, etc.).

- **Linux clone()**
  - Constants with flags in clone()
    - CLONE_NEWNET (2)
      - A physical network device can live in exactly one network namespace. A virtual network device ("veth") pair provides a pipe-like abstraction that can be used to create tunnels between network namespaces, and can be used to create a bridge to a physical network device in another namespace.
      - When a network namespace is freed (i.e., when the last process in the namespace terminates), its physical network devices are moved back to the initial network namespace (not to the parent of the process).
      - Use of this flag requires: a kernel configured with the CONFIG_NET_NS option and that the process be privileged (CAP_SYS_ADMIN)

Linux clone()

- Constants with flags in clone()
  - CLONE_THREAD
    - If CLONE_THREAD is set, the child is placed in the same thread group as the calling process.
      - Here the term "thread" is used to refer to the processes within a thread group.
    - Thread groups were a feature added in Linux 2.4 to support the POSIX threads notion of a set of threads that share a single PID. Internally, this shared PID is the so-called thread group identifier (TGID) for the thread group. Since Linux 2.4, calls to getpid(2) return the TGID of the caller.
    - The threads within a group can be distinguished by their (system-wide) unique thread IDs (TID). A new thread's TID is available as the function result returned to the caller of clone(), and a thread can obtain its own TID using gettid(2).

Linux clone()

- Constants with flags in clone()
  - CLONE_THREAD (2)
    - A new thread created with CLONE_THREAD has the same parent process as the caller of clone() (i.e., like CLONE_PARENT), so that calls to getppid(2) return the same value for all of the threads in a thread group.
    - When a CLONE_THREAD thread terminates, the thread that created it using clone() is not sent a SIGCHLD (or other termination) signal; nor can the status of such a thread be obtained using wait(2). (The thread is said to be *detached*.)
    - After all of the threads in a thread group terminate the parent process of the thread group is sent a SIGCHLD (or other termination) signal.
    - When a call is made to clone() without specifying CLONE_THREAD, then the resulting thread is placed in a new thread group whose TGID is the same as the thread's TID. This thread is the *leader* of the new thread group.

Linux clone()

- Constants with flags in clone()
  - CLONE_THREAD (3)
    - If any of the threads in a thread group performs an execve(2), then all threads other than the thread group leader are terminated, and the new program is executed in the thread group leader.
    - If one of the threads in a thread group creates a child using fork(2), then any thread in the group can wait(2) for that child.
    - Since Linux 2.5.35, *flags* must also include CLONE_SIGHAND if CLONE_THREAD is specified.
    - Signals may be sent to a thread group as a whole (i.e., a TGID) using kill(2), or to a specific thread (i.e., TID) using tgkill(2).
    - Signal dispositions and actions are process-wide: if an unhandled signal is delivered to a thread, then it will affect (terminate, stop, continue, be ignored in) all members of the thread group.

Linux clone()

- Constants with flags in clone()
  - CLONE_THREAD (4)
    - Each thread has its own signal mask, as set by sigprocmask(2), but signals can be pending either: for the whole process (i.e., deliverable to any member of the thread group), when sent with kill(2); or for an individual thread, when sent with tgkill(2).
    - A call to sigpending(2) returns a signal set that is the union of the signals pending for the whole process and the signals that are pending for the calling thread.
    - If kill(2) is used to send a signal to a thread group, and the thread group has installed a handler for the signal, then the handler will be invoked in exactly one, arbitrarily selected member of the thread group that has not blocked the signal. If multiple threads in a group are waiting to accept the same signal using sigwaitinfo(2), the kernel will arbitrarily select one of these threads to receive a signal sent using kill(2).

Linux clone()

- Constants with flags in clone()
  - *CLONE_STOPPED
    - If CLONE_STOPPED is set, then the child is initially stopped (as though it was sent a SIGSTOP signal), and must be resumed by sending it a SIGCONT signal.
    - This flag was *deprecated* from Linux 2.6.25 onward, and was *removed* altogether in Linux 2.6.38.
  - *CLONE_PID
    - If CLONE_PID is set, the child process is created with the same process ID as the calling process. This is good for hacking the system, but otherwise of not much use. Since 2.3.21 this flag can be specified only by the system boot process (PID 0). It disappeared in Linux 2.5.16.

Examples

alg.14-6-clone-demo.c: clone() with VM and VFORK (1)

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sched.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/syscall.h>
#include <unistd.h>
#define gettid() syscall(__NR_gettid)
  /* wrap the system call syscall(__NR_gettid), __NR_gettid = 224 */
#define gettidv2() syscall(SYS_gettid) /* a traditional wrapper */
#define STACK_SIZE 1024*1024 /* 1Mib. question: what is the upperbound of STACK_SIZE */

static int child_func1(void *arg)
{
    char *chdbuf = (char*)arg; /* type casting */
    printf("child_func1 read buf: %s\n", chdbuf);
    sprintf(chdbuf, "I am child_func1, my tid = %ld, pid = %d", gettid(), getpid());
    printf("child_func1 set buf: %s\n", chdbuf);
    printf("child_func1 sleeping and then exists ...\n");
    sleep(1);

    return 0;
}
```

Examples

alg.14-6-clone-demo.c: clone() with VM and VFORK (2)

```
static int child_func2(void *arg)
{
    char *chdbuf = (char*)arg; /* type casting */
    printf("child_func2 read buf: %s\n", chdbuf);
    sprintf(chdbuf, "I am child_func2, my tid = %ld, pid = %d", gettid(), getpid());
    printf("child_func2 set buf: %s\n", chdbuf);
    printf("child_func2 sleeping and then exists ...\n");
    sleep(1);

    return 0;
}

int main(int argc,char **argv)
{
    char *stack1 = malloc(STACK_SIZE*sizeof(char));
        /* allocating from heap, safer than stack1[STACK_SIZE] */
    char *stack2 = malloc(STACK_SIZE*sizeof(char));
    pid_t chdtid1, chdtid2;
    unsigned long flags = 0;
    char buf[100]; /* a global variable has the same behavior */

    if(!stack1 || !stack2) {
        perror("malloc()");
        exit(1);
    }
```

Examples

alg.14-6-clone-demo.c: clone() with VM and VFORK (3)

```
  /* set CLONE flags */
if((argc > 1) && (!strcmp(argv[1], "vm"))) {
    flags |= CLONE_VM;
}
if((argc > 2) && (!strcmp(argv[2], "vfork"))) {
    flags |= CLONE_VFORK;
}

sprintf(buf,"I am parent, my pid = %d", getpid());
printf("parent set buf: %s\n", buf);
printf("parrent clone ...\n");

  /* creat child thread, top of child stack is stack+STACK_SIZE */
chdtid1 = clone(child_func1, stack1 + STACK_SIZE, flags | SIGCHLD, buf);
    /* what happened without SIGCHLD */
if(chdtid1 == -1) {
    perror("clone1()");
    exit(1);
}

chdtid2 = clone(child_func2, stack2 + STACK_SIZE, flags | SIGCHLD, buf);
if(chdtid2 == -1) {
    perror("clone2()");
    exit(1);
}
```

What will happen if clone() with CLONE_VM but without SIGCHLD?

Examples

alg.14-6-clone-demo.c: clone() with VM and VFORK (4)

```
    printf("parent waiting ... \n");

    int status = 0;
    if(waitpid(-1, &status, 0) == -1) { /* wait for any child existing, may leave some child
defunct */
        perror("wait()");
    }

//waitpid(chdtid1, &status, 0);
//waitpid(chdtid2, &status, 0);

    sleep(2);

    printf("parent read buf: %s\n", buf);

    system("ps");

    free(stack1);
    free(stack2);
    stack1 = NULL;
    stack2 = NULL;

    return 0;
}
```

Linux clone()

Examples

- alg.14-6-clone-demo.c: clone() with VM and VFORK (4)

```
isscgy@ubuntu:/mnt/os-2020$ gcc alg.14-6-clone-demo.c
isscgy@ubuntu:/mnt/os-2020$ ./a.out
parent set buf: I am parent, my pid = 48233
parrent clone ...
parent waiting ...
child_func1 read buf: I am parent, my pid = 48233
child_func2 read buf: I am parent, my pid = 48233
child_func2 set buf: I am child_func2, my tid = 48235, pid = 48235
child_func1 set buf: I am child_func1, my tid = 48234, pid = 48234
child_func2 sleeping and then exists ...
child_func1 sleeping and then exists ...
parent read buf: I am parent, my pid = 48233
   PID TTY          TIME CMD
  1957 pts/0    00:00:00 bash
 47134 pts/0    00:00:00 bash
 48233 pts/0    00:00:00 a.out
 48235 pts/0    00:00:00 a.out <defunct>
 48236 pts/0    00:00:00 sh
 48237 pts/0    00:00:00 ps
```

Every task has its own memory space.

Examples

- alg.14-6-clone-demo.c: clone() with VM and VFORK (4)

```
isscgy@ubuntu:/mnt/os-2020$ gcc alg.14-6-clone-demo.c
isscgy@ubuntu:/mnt/os-2020$ ./a.out
parent set buf: I am parent, my pid = 48233
parrent clone ...
parent waiting ...
child_func1 read buf:                          48233
child_func2 read buf: I am parent, my pid = 48233
child_func2 set buf: I am child_func2, my tid = 48235, pid = 48235
child_func1 set buf: I am child_func1, my tid = 48234, pid = 48234
child_func2 sleeping and then exists ...
child_func1 sleeping and then exists ...
parent read buf: I am parent, my pid = 48233
   PID TTY          TIME CMD
  1957 pts/0    00:00:00 bash
 47134 pts/0    00:00:00 bash
 48233 pts/0    00:00:00 a.out
 48235 pts/0    00:00:00 a.out <defunct>
 48236 pts/0    00:00:00 sh
 48237 pts/0    00:00:00 ps
```

Parent and his children progress asynchronously

Examples

- alg.14-6-clone-demo.c: clone() with VM and VFORK (4)

```
isscgy@ubuntu:/mnt/os-2020$ ./a.out vm
parent set buf: I am parent, my pid = 48238
parrent clone ...
parent waiting ...
parent waiting ...
child_func1 read buf: I am parent, my pid = 48238
child_func2 read buf: I am parent, my pid = 48238
child_func2 set buf: I am child_func2, my tid = 48240, pid = 48240
child_func2 set buf: I am child_func2, my tid = 48240, pid = 48240
child_func1 set buf: I am child_func1, my tid = 48239, pid = 48239
child_func2 sleeping and then exists ...
child_func1 sleeping and then exists ...
parent read buf: I am child_func1, my tid = 48239, pid = 48239
   PID TTY          TIME CMD
  1957 pts/0    00:00:00 bash
 47134 pts/0    00:00:00 bash
 48238 pts/0    00:00:00 a.out
 48240 pts/0    00:00:00 a.out <defunct>
 48241 pts/0    00:00:00 sh
 48242 pts/0    00:00:00 ps
```

Tasks share memory space.

Examples

- alg.14-6-clone-demo.c: clone() with VM and VFORK (4)

```
isscgy@ubuntu:/mnt/os-2020$ ./a.out vm
parent set buf: I am parent, my pid = 48238
parrent clone ...
parent waiting ...
parent waiting ...
child_func1 read buf: I am parent, my pid = 48238
child_func2 read buf: I am parent, my pid = 48238
child_func2 set buf: I am child_func2, my tid = 48240, pid = 48240
child_func2 set buf: I am child_func2, my tid = 48240, pid = 48240
child_func1 set buf: I am child_func1, my tid = 48239, pid = 48239
child_func2 sleeping and then exists ...
child_func1 sleeping and then exists ...
parent read buf: I am child_func1, my tid = 48239, pid = 48239
   PID TTY          TIME CMD
  1957 pts/0    00:00:00 bash
 47134 pts/0    00:00:00 bash
 48238 pts/0    00:00:00 a.out
 48240 pts/0    00:00:00 a.out <defunct>
 48241 pts/0    00:00:00 sh
 48242 pts/0    00:00:00 ps
```

Parent and his children progress asynchronously

Examples

- alg.14-6-clone-demo.c: clone() with VM and VFORK (4)

```
isscgy@ubuntu:/mnt/os-2020$ ./a.out vm vfork
parent set buf: I am parent, my pid = 48243
parrent clone ...
child_func1 read buf: I am parent, my pid = 48243
child_func1 set buf: I am child_func1, my tid = 48244, pid = 48244
child_func1 sleeping and then exists ...
child_func2 re[illegible]ld_func1, my tid = 48244, pid = 48244
child_func2 se[illegible]ld_func2, my tid = 48245, pid = 48245
child_func2 sleeping and then exists ...
parent waiting ...
parent read buf: I am child_func2, my tid = 48245, pid = 48245
   PID TTY          TIME CMD
  1957 pts/0    00:00:00 bash
 47134 pts/0    00:00:00 bash
 48243 pts/0    00:00:00 a.out
 48245 pts/0    00:00:00 a.out <defunct>
 48246 pts/0    00:00:00 sh
 48247 pts/0    00:00:00 ps
```

Parent suspended

}

Examples

- Alg.14-7-clone-stack.c: test the upper bound of clone stack (1)

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <sched.h>
#include <sys/wait.h>
#include <unistd.h>
#include <sys/syscall.h>

#define gettid() syscall(__NR_gettid)
#define STACK_SIZE (524288-10000)*4096 /* 2^19 = 524288 */

static int test(void *arg)
{
    static int i = 0;
    char buffer[1024];
    if (i == 0) {
        printf("test: my ptd = %d, tid = %ld, ppid = %d\n", getpid(), gettid(), getppid());
        printf("\niteration = %8d", i);
    }
    printf("\b\b\b\b\b\b\b\b%8d", i);
    i++;
    test(arg); /* recursive calling */
}
```

Examples

- Alg.14-7-clone-stack.c: test the upper bound of clone stack (2)

```
int main(int argc,char **argv)
{
    char *stack = malloc(STACK_SIZE); /* allocating from heap */
    pid_t chdtid;
    char buf[40];
    if(!stack) {
        perror("malloc()");
        exit(1);
    }
    unsigned long flags = 0;
    chdtid = clone(test, stack + STACK_SIZE, flags | SIGCHLD, buf);
    if(chdtid == -1)
        perror("clone()");
    printf("\nmain: my pid = %d, I'm waiting for cloned child, his tid = %d\n", getpid(),
chdtid);

    int status = 0;
    int ret;
    ret = waitpid(-1, &status, 0); /* wait for any child existing */
    if(ret == -1)
        perror("waitpid()");
    sleep(2);
    printf("\nmain: my pid = %d, waitpid returns = %d\n", getpid(), ret);
    free(stack);
    stack = NULL;
    return 0;
}
```

Examples

- Alg.14-7-clone-stack.c: test the upper bound of clone stack (2)

```
int main(int argc,char **argv)
{
    char *stack = malloc(STACK_SIZE); /* allocating from heap */
    pid_t chdtid;
```

```
isscgy@ubuntu:/mnt/os-2020$ gcc alg.14-7-clone-stack.c
isscgy@ubuntu:/mnt/os-2020$ ./a.out

main: my pid = 48422, I'm waiting for cloned child, his tid = 48423
test: my ptd = 48423, tid = 48423, ppid = 48422

iteration =  1936125
main: my pid = 48422, waitpid returns = 48423
isscgy@ubuntu:/mnt/os-2020$
```

```
    int status = 0;
    int ret;
    ret = waitpid(-1, &status, 0); /* wait for any child existing */
    if(ret == -1)
        perror("waitpid()");
    sleep(2);
    printf("\nmain: my pid = %d, waitpid returns = %d\n", getpid(), ret);
    free(stack);
    stack = NULL;
    return 0;
}
```

Windows Threads

- Windows implements the Windows API – primary API for Win 98, Win NT, Win 2000, Win XP, and Win 7.
- Implements the one-to-one mapping, kernel-level.
- Each thread contains:
  - A thread id.
  - Register set representing state of processor.
  - Separate user and kernel stacks for when thread runs in user mode or kernel mode.
  - Private data storage area used by run-time libraries and dynamic link libraries (DLLs).
- The register set, stacks, and private storage area are known as the context of the thread.

Windows Threads

- The primary data structures of a thread include:
  - ETHREAD (executive thread block)
    - includes pointer to process to which thread belongs and to KTHREAD, in kernel space.
  - KTHREAD (kernel thread block)
    - scheduling and synchronization info, kernel-mode stack, pointer to TEB, in kernel space.
  - TEB (thread environment block)
    - thread id, user-mode stack, thread-local storage, in user space.

Windows Threads

- The primary data structures of a thread