

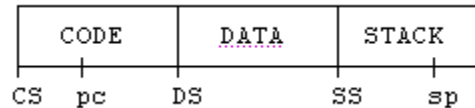
## Chapter 5. User Mode and System Calls

**Abstract:** Chapter 5 covers user mode processes and system calls. It explains the process images in user and kernel modes, the mechanism of transition between user and kernel modes and how to implement the transition. Then it shows how to create processes with user mode images and let the processes return to execute their images in user mode. Based on these, it shows how to implement simple system calls, which allow user mode processes to enter kernel, execute kernel functions and return to user mode. Then it implements other system calls for process management. These include fork, exec and the advanced technique of vfork. With fork-exec, it shows how to start up MTX with an INIT process, which forks a rudimentary sh process to run user commands. Then it explains the concepts and advantages of threads and extends the process model to implement threads support in the MTX kernel. In addition, it demonstrates threads applications by concurrent programs and shows how to synchronize threads executions to prevent race conditions.

In Chapter 4, we developed a simple OS kernel for process management. The simple kernel supports dynamic process creation, process termination, process synchronization by sleep/wakeup operations, wait for child process termination and priority-based process scheduling. In MTX4.5, all processes run in kernel mode since they execute in the same address space of the MTX kernel. In a real operating system, processes may execute in two different modes; kernel mode and user mode. In this chapter, we shall extend the simple MTX kernel to support process executions in user mode. First, we explain the difference between the address spaces of kernel and user modes, the mechanism of transition between user and kernel modes, and how to implement the transition. Then we show how to create processes with user mode images and let them return to execute images in user mode. Based on these, we implement simple system calls, which allow user mode processes to enter kernel mode, execute kernel functions and return to user mode. Then we implement other system call functions for process management. These include fork, exec and the advanced technique of vfork [Goldt et al., 1995]. This would make the MTX kernel to have similar capability as the Unix kernel for process management. In addition to the fork-exec paradigm of Unix, we also point out other alternative schemes, such as the create and attach operations in MVS [IBM MVS]. Then we discuss the concepts and advantages of threads [POSIX, 1995] and extend the process model to implement threads support in the MTX kernel. In addition, we also demonstrate threads applications by concurrent programs and show how to synchronize threads executions to prevent race conditions. In the Problem section, we point out variations to the kernel design and encourage the reader to explore alternative ways to implement the kernel functions.

### 5.1. Process Execution Image

The execution image of a process consists of three logical segments; Code, Data and Stack, as shown in Figure 5.1.



**Figure 5.1. Process Execution Image**

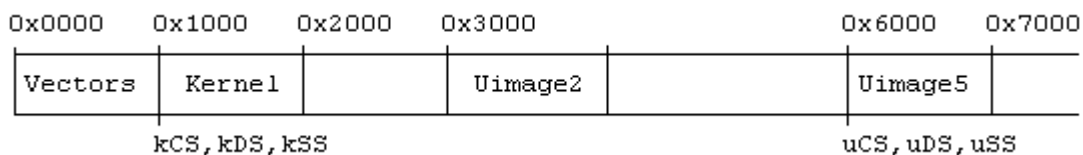
In theory, all the segments can be independent, each in a different memory area, as long as they are pointed by the CPU's segment registers. In practice, some of the segments may coincide. For example, in the single-segment memory model, all segments are the same. During execution, the CPU's CS, DS and SS all point to the same segment of the execution image. In the separate I&D space memory model, CS points to the Code segment, but DS and SS point to the combined Data and Stack segment. For the sake of simplicity, we shall assume the single-segment memory model first. Each process image has only one segment and the segment size is 64 KB. Process images with separate I&D spaces and variable sizes will be considered later in Chapter 7 when we discuss memory management.

## 5.2. Kernel and User Modes

From now on we shall assume that a process may execute in two different modes; kernel mode and user mode, denoted by Kmode and Umode for short. While in Kmode, all processes share the same Code and Data of the OS kernel, but each process has its own kernel mode stack (in the PROC structure), which contains the execution context of the process while in Kmode. When a process gives up CPU, it saves its dynamic context (CPU registers) in kstack. When a process becomes running again, it restores the saved context from kstack. In Umode, process images are in general all different. Each process has a separate Umode memory area containing the user mode code, data and stack, denoted by Ucode, Udata and Ustack, of the process. For ease of discussion, we begin with the following assumptions.

- (1). The MTX kernel runs in the segment 0x1000.
- (2). The system has 9 PROC structures, P0 to P8. P0 always runs in Kmode.
- (3). Only P1 to P8 may run in Umode, each has a distinct 64 KB Umode image in the segment  $(pid+1) * 0x1000$ , e.g. P1 in 0x2000, P2 in 0x3000, P8 in 0x9000, etc.

The fixed segment memory assignment of process images is just for ease of discussion. It will be removed later when we implement memory management. Next, assume that the process P5 is running in Umode now, P2 is READY but not running. Figure 5.2 shows the current memory map of the MTX system.



**Figure 5.2. Processes with Umode Images**

Since P5 is running in Umode, the CPU's CS, DS, SS registers must all point to P5's Umode image at 0x6000. P5's ustack is in the upper region of Uimage5. When P5 enters Kmode, it will execute the Kernel image in the segment 0x1000. The CPU's CS, DS, SS registers must be changed to point to 0x1000, as shown by the kCS, kDS, kSS in the figure. In order for P5 to run in kernel, the stack pointer must also be changed to point to P5's kstack. Naturally, P5 must save its Umode segment registers, uCS, uDS, uSS, and usp if it intends to return to Uimage5 later.

Assume that, while in Kmode, P5 switches to P2, which may return to its Umode image at 0x3000. If so, P2 must change CPU's segment registers to its own uCS, uDS, uSS, all of which must point to 0x3000. When P2 runs in Umode, its ustack is in the upper region of Uimage2. Similarly, the reader may deduce what would happen if P2 enters Kmode to switch to another process, etc.

### 5.3. Transition between User and Kernel modes

In an OS, a process migrates between Umode and Kmode many times during its lifetime. Although every process begins in Kmode, we shall assume that a process is already executing in Umode. This sounds like another chicken-egg problem, but we can handle it easily. A process in Umode will enter Kmode if one of the following events occurs:

- . Exceptions : also called traps, such as illegal instruction, invalid address, etc.
- . Interrupts : timer interrupts, device I/O completion, etc.
- . System calls : INT n (or equivalent instructions on other CPUs).

Exceptions and interrupts will be covered in later chapters. Here, we only consider system calls. System call, or syscall for short, is a mechanism which allows a process in Umode to enter Kmode to execute kernel functions. Syscalls are not ordinary function calls because they involve CPU operating in different modes (if so equipped) and executing in different address spaces. However, once the linkage is set up, syscalls can be used as if they were ordinary function calls. Assume that there are N kernel functions, each corresponds to a call number  $n = 0, 1, \dots, N-1$ , e.g.

Call#	Kernel Function	
-----	-----	
0	kgetpid()	// get process pid
1	kfork()	// fork a child process
2	kexec()	// change Umode image
3	kwait()	// wait for ZOMBIE child process
.....		
6	kexit()	// terminate

where the prefix k of the function names emphasizes that they are functions in the OS kernel. A Umode process may use

```
int r = syscall(call#, parm1, parm2, .... );
```

to enter Kmode to execute the corresponding kernel function, passing parameters to the kernel function as needed. When the kernel function finishes, the process returns to

Umode with the desired results and a return value. For most syscalls, a 0 return value mean success and -1 means failure.

### 5.3.1. System Calls

Assuming 4 parameters, the implementation of syscall() is shown below.

```

=====
!  int syscall(int a,b,c,d); issue INT 80 to enter Kernel
=====
_syscall:
        INT n    <==== This is the magic wand!
returnHERE:  ret

```

On the Intel x86 CPU, syscall is implemented by the INT n instruction, where n is a byte value (0 to 255). Although we may use different INT n to implement different syscalls, it suffices to use only one number: INT 80, since the parameter a represents the syscall number. The choice of INT 80 is quite arbitrary. We may choose any other number, e.g. 0x80, as long as it is not used as IRQ of the interrupt hardware or by BIOS. When the CPU executes INT 80, it does the following.

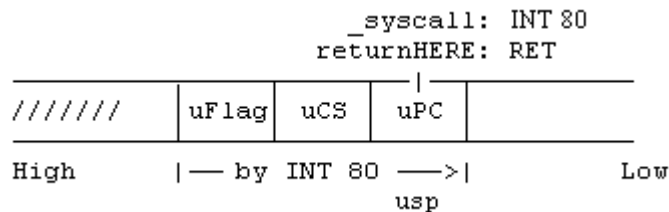
PUSH : push flag register, clear flag register's T-bit and I-bit; push uCS and uPC.

LOAD: load (PC, CS) with contents of (4\*80, 4\*80+2) = (\_int80h, KCS).

HANDLER: continue execution from the loaded (PC, CS) = (\_int80h, KCS).

Since these operations are crucial to understanding both system call and interrupts processing in later chapters, we shall explain them in more detail.

PUSH: INT n causes the CPU to push the current uFlag, uCS, uPC into stack. On most other machines, a special instruction like INT n causes the CPU to enter a separate Kmode and switches stack to Kmode stack automatically. The Intel x86 CPU in 16-bit real mode does not have a separate Kmode or a separate Kmode stack pointer. After executing INT 80, it only switches the execution point from (uPC, uCS) to (kPC, kCS), which changes the code segment from UCode to KCode. All other segments (DS, SS, ES) and CPU registers are still those in Umode. Thus, when CPU first enters the int80h() handler function, the stack is still the process ustack in the uSS segment. It contains the saved uFlag, uCS, uPC at the top, where uPC points at the address of returnHERE in \_syscall, as shown in Figure 5.3.



**Figure 5.3. Stack Contents by INT 80**

Corresponding to INT n, the instruction IRET pops three items off the current stack into CPU's PC, CS, Flag registers, in that order. It is used by interrupt handlers to return to the original point of interruption.

LOAD: For the x86 CPU in 16-bit real mode, the lowest 1KB area of physical memory is dedicated to 256 interrupt vectors. Each interrupt vector area contains a pair of (PC, CS), which points to the entry point of an interrupt handler. After saving uFlag, uCS and uPC of the interrupted point, the CPU turns off the T (trace) and I (Interrupt Mask) bits in the flag register to disable trace trap and mask out interrupts. Then it loads (PC, CS) with the contents of interrupt vector 80 as the new execution point. The interrupt vector 80 area must be initialized before executing INT 80, as shown below.

```
set_vector(80, (int)int80h); // int80h() is _int80h: in ts.s file
int set_vector(int vector, int handler)
{
    put_word(handler, 0x0000, vector*4); // KPC points to handler
    put_word(0x1000, 0x0000, vector*4+2); // KCS segment=0x1000
}
```

### 5.3.2. System Call Interrupt Handler

After loading the vector 80 contents into (PC, CS) registers, the CPU executes int80h() in the code segment of the MTX kernel. int80h() is the entry point of INT 80 interrupt handler in assembly code, which is shown below.

```
HANDLER: INT80 interrupt handler
|=====
| int80h() is the entry point of INT 80 interrupts
|=====
_int80h:
! (1). SAVE Umode registers into ustack, save uSS,uSP into running PROC
! Accordingly, we modify the PROC structure as follows.
!     typedef struct proc{
!         struct proc *next;
!         int *ksp;
!         int uss, usp; // ADD uss, usp at byte offsets 4, 6
!         Other PROC fields are the same as before
!     } PROC;
!     int procSzie = sizeof(PROC): // a global used in assembly code
!     When a process enters Kmode, saves (uSS,uSP) to PROC.(uss,usp)
! (2). Set stack pointer sp to running PROC's kstack HIGH END
!     Then call handler function in C
! (3). RETURN to Umode
! Details of the steps (1) to (3) are shown below.
! ***** SAVE U mode registers *****
KSAVE:
    push ax    ! save all Umode registers into ustack
    push bx
    push cx
    push dx
    push bp
    push si
    push di
    push es
    push ds
! ustack contains: |uflag,uCS,uPC|ax,bx,cx,dx,bp,si,di,ues,uds|
!                                                         usp
! change DS to KDS in order to access data in Kernel space
```

```

        mov ax,cs          ! assume one-segment kernel, change DS to kDS
        mov ds,ax          ! let DS=CS = 0x1000 = kernel DS
USS = 4          ! offsets of uss, usp in PROC
USP = 6
! All variables are now offsets relative to DS=0x1000 of Kernel space
! Save running proc's Umode (uSS, uSP) into its PROC.(uss, usp)
        mov bx,_running ! bx points to running PROC in K space
        mov USS(bx),ss  ! save uSS in proc.uss
        mov USP(bx),sp  ! save uSP in proc.usp
! change ES,SS to Kernel segment 0x1000
        mov ax,ds       ! CPU must mov segments this way!
        mov es,ax
        mov ss,ax       ! SS is now KSS = 0x1000
! switch running proc's stack from U space to K space.
        mov sp,bx       ! sp points at running PROC
        add sp,procSize ! sp -> HIGH END of running's kstack
! We are now completely in K space, stack=running proc's EMPTY kstack
! ***** CALL handler function in C *****
        call_kcint      ! call kcint() in C;
! ***** RETURN TO Umode *****
_goUmode:
        cli             ! mask out interrupts
        mov bx,_running ! bx -> running PROC
        mov ax,USS(bx)
        mov ss,ax       ! restore uSS
        mov sp,USP(bx)  ! restore uSP
        pop ds
        pop es
        pop di
        pop si
        pop bp
        pop dx
        pop cx
        pop bx
        pop ax          ! NOTE: return value must be in AX in ustack
        iret

```

The C handler function, `kcint()`, actually handles the syscall. When the C handler function finishes, the process returns to execute the assembly code `_goUmode`. It first restores the Umode stack from running PROC.(`uss, usp`). Then it restores the saved Umode registers, followed by IRET, causing the process to return to the interrupted point in Umode. Since `_goUmode` is a global symbol, a process in Kmode may call `goUmode()` directly to return to Umode.

### 5.3.3. Enter and Exit Kernel Mode

The assembly code `_int80h` and `_goUmode` are the keys to understanding transitions between Umode and Kmode. Logically, `int80h()` and `goUmode()` are similar to the SAVE and RESUME operations of `tswitch()` in Kmode. The following compares their similarity and differences.

```

tswitch() in Kmode:
{
    SAVE : save CPU registers in kstack; save kSP in PROC;

```

```

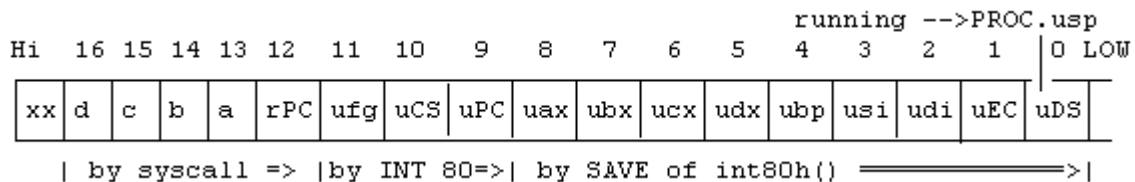
    FIND : // call scheduler() to find next running PROC;
    RESUME: restore kSP from running PROC and CPU regs from kstack;
    RET   : return to where it called tswitch() earlier;
}
int80h()
{
    1. save CPU registers in ustack;
    2. switch CPU from Umode to Kmode, save uSS and uSP in PROC;
       change stack to running PROC's EMPTY kstack;
    3. call handler function kcinth() in C;
}
// kernel may switch running to a different PROC before goUmode()
goUmode()
{
    1. restore Umode uSS and uSP from running PROC;
    2. restore saved CPU registers from ustack;
    3. iret back to Umode;
}

```

The major difference is 2 in `int80h()`, which changes the CPU's execution environment from Umode to Kmode and saves Umode's `uSS` and `uSP` in the running PROC. In most CPUs designed for Kmode and Umode operations, the switch is automatic. When such a CPU executes the `syscall` instruction (or accepts an interrupt), it automatically switches to Kmode and saves the interrupt context in Kmode stack. Since the x86 CPU in real mode does not have this capability, we have to coerce it to do the switch manually, which is both a curse and blessing. It's a curse because we have to do the extra work. It's a blessing because it allows the reader to better understand how CPU switches mode and execution environment. `_int80h` and `_goUmode()` are the entry and exit code of the `syscall` handler. Since `syscall` is just a special kind of interrupt, the same entry and exit code can also be used to handle other kinds of interrupts and exceptions. We shall show this later in Chapters 8 and 9 when we discuss interrupt and exception processing.

#### 5.3.4. System Call Handler Function in C

`kcinth()` is the `syscall` handler function in C. The parameters used in `syscall(a,b,c,d)` are in the process `ustack`, which contains the entries as shown in Figure 5.4.



**Figure 5.4. Process System Call Stack Contents**

The Umode segment is saved in `PROC.uss`, and `usp` is saved in `PROC.usp`. Using the inter-segment copying functions, `get_word()/put_word()`, we can access the process `ustack`. For example, we can get the `syscall` number `a` (at index 13) by

```
int ka = get_word(running->uss, running->usp + 2*13);
```

Similarly, we can get other syscall parameters from the ustack. Based on the call number a, we can route the syscall to a corresponding kernel function, passing parameters b, c, d to it as needed. The kernel function actually handles the syscall. As an example, assume that the syscall number 0 is to get the process pid. The call is routed to

```
int r = kgetpid(){ return running->pid;}
```

which simply returns the running process pid. Likewise, we can write to the ustack of a process to change its contents. In particular, we can change the saved uax (at index 8) as the return value to Umode by `put_word(r, running->uss, running->usp + 2*8);`

### 5.3.5. Return to User Mode

Each kernel function returns a value back to Umode, except `kexit()`, which never returns, and `kexec()`, which returns to a different image if the operation succeeds. The return value is carried in the CPU's AX register. Since `goUmode()` pops the saved uax from ustack, we must fix up the saved uax with the desired return value before executing `goUmode()`. This can be done in the C handler function `kcinth()` before it returns to execute `goUmode()`.

The above describes the control flow of syscalls. It shows the execution locus of a process when it issues a syscall to enter kernel, executes kernel functions and returns to Umode with a return value. Syscalls are the main mechanism in support of kernel mode and user mode operations. Implementation of the syscall mechanism is a major step in the development of an OS kernel. In the following, we shall show how to implement syscalls in MTX. This should allow the reader to better understand the internal operations of an operating system kernel.

## 5.4. User Mode Programs

In order to issue syscalls, we need a Umode image file, which is a binary executable to be executed by a process in Umode.

### 5.4.1. Develop User Program Programs

The following shows a simple Umode program. As usual, it consists of a `u.s` file in BCC assembly and a `u1.c` file in C.

```
!===== u.s file of Umode image =====
                .globl _main,_syscall,_exit,_getcs
start:          call _main
! if main() returns, syscall exit(0) to terminate in MTX kernel
                push #0                ! push exitValue 0
                call _exit             ! call exit(value) function in Umode
! int syscall(a,b,c,d) from C code
_syscall:       int 80
                ret
_getcs:         mov ax,cs              ! getcs() return CS segment register
                ret

/***** u1.c file of Umode image *****/
```



```

main()
{
    int pid = getpid();
    printf("I am proc %d in Umode: running segment=%x\n",pid,getcs());
    printf("Enter a key : "); getc();
    printf("proc %d syscall to kernel to die\n", pid);
    exit(0);
}
int getpid()                // assume : getpid() call# = 0
{ return syscall(0, 0, 0, 0); }

int exit(int exitValue)     // assume : exit() call# = 6
{ syscall(6, exitValue, 0, 0); }

```

The `u.s` file is necessary because we can only issue syscalls by INT 80 in assembly. It also serves as the entry point of Umode programs. As will be seen shortly, when a Umode program begins execution, the CPU's segment registers are already set up by the kernel code and it already has a `ustack`. So upon entry, `u.s` simply calls `main()` in C.

A Umode program can only do general computations. Any operation not available in Umode must be done by syscalls to the OS kernel. For example, both `getpid()` and `exit()` are syscalls. This is because a process can only get its pid from kernel space and it must terminate in kernel. Each syscall has an interface function, which issues an actual syscall to the kernel. For convenience, syscall interface functions are implemented in a single file, `ucode.c`, which is shared by all Umode programs. Usually, they are precompiled as part of the system linking library. Umode programs may call syscall interface functions, such as `getpid()`, `exit()`, etc. as ordinary function calls.

As of now, the MTX kernel does not yet have its own device drivers. All I/O operations in MTX are based on BIOS. I/O functions, such as `getc()`, `putc()`, `gets()`, `printf()`, are precompiled object code in a `mtxlib` library, which is used by both the MTX kernel and Umode programs. Therefore, `getc()` and `putc()` in Umode are also BIOS calls. Strictly speaking, Umode programs should not be able to do I/O directly. Basic Umode I/O, e.g. `getc()` and `putc()`, should also be syscalls. This is left as an exercise. As before, use BCC to generate a binary executable (with header, which is needed by the loader), as in

```

as86 -o u.o u.s
bcc -c -ansi u1.c
ld86 -o u1 u.o u1.o mtxlib /usr/lib/bcc/libc.a
mount -o loop mtximage /mnt; cp u1 /mnt/bin; umount /mnt

```

The last command line copies `u1` to `/bin/u1` in a bootable MTX system image.

### 5.4.2. Program Loader

Executable `a.out` files generated by BCC have a 32-bit header containing 8 long values.

```

struct header{
    u32 ID_space: // 0x4100301:combined I&D|0x4200301:separate I&D
    u32 magic_number; // 0x00000020
    u32 tsize;        // code section size in bytes
    u32 dsize;        // initialized data section size in bytes
    u32 bsize;        // bss section size in bytes
}

```

```

    u32 zero;           // 0
    u32 total_size;     // total memory size, including heap
    u32 symbolTable_size; // only if symbol table is present
}

```

A loader is a program which loads a binary executable file into memory for execution. In a real OS kernel with file system support, the loader typically uses the kernel's internal `open()` function to open the image file for read. Then it uses the kernel's internal `read()` function to load the image file into memory. We shall show this later when we add file system support to the MTX kernel. In the meantime, we shall modify the MTX booter as a loader. In this case, the loader is almost the same as a booter, except that it is not a standalone program but a callable function in kernel. The loader's algorithm is

```

/***** Algorithm of MTX Loader *****/
int load(char *filename, u16 segment)
{
    1. find the inode of filename; return 0 if fails;
    2. read file header to get tsize, dsize and bsize;
    3. load [code|data] sections of filename to memory segment;
    4. clear bss section of loaded image to 0;
    5. return 1 for success;
}

```

## 5.5. Create Process with User Mode Image

In the MTX kernel, we modify `kfork()` to `kfork(char *filename)`, which creates a new process and loads filename as its Umode image. The following show the modified `kfork` algorithm.

```

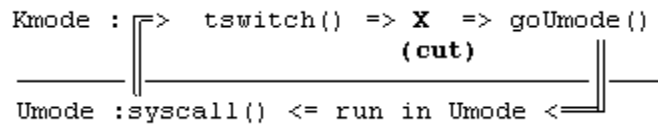
PROC *kfork(char *filename)
{
    (1). creat a child PROC ready to run from body() in Kmode
    (2). segment=(child pid + 1)*0x1000; //child Umode segment
        if (filename){
    (3).    load filename to child segment as its Umode image;
    (4).    set up child's ustack for it to return to Umode to execute
           the loaded image;
        }
    (5). return child PROC pointer;
}

```

In the modified `kfork()`, Step (1) is the same as before. The MTX kernel has 9 PROCs but only P1 to P8 may run in Umode. The Umode segment of each process is statically assigned in Step (2) as  $(pid+1)*0x1000$ , so that each process runs in a unique Umode segment. In Step (3), it calls `load("/bin/u1", segment)`, which loads `/bin/u1` into the Umode segment of the new process and clears the bss section in the Umode image to 0. Step (4) is most crucial, so we shall explain it in detail.

## 5.6. Initialize Process User Mode Stack

Our objective here is to set up the ustack of a new process for it to return to Umode to execute the loaded image. To do this, we again think of each process as running in a perpetual cycle, as shown in Figure 5.5.



**Figure 5.5. Process Execution Cycle**

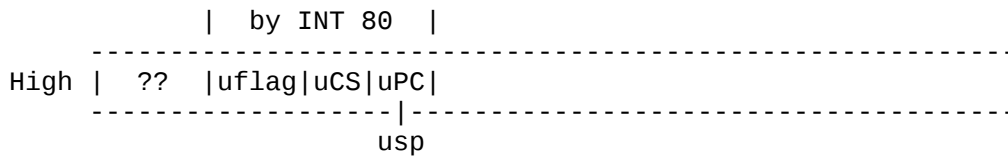
In Figure 5.5, a process in Umode does a syscall to enter Kmode, executes in Kmode until it calls tswitch() to give up CPU and stays in the readyQueue until it runs again. Then it executes goUmode() to return to Umode and repeats the cycle. We can make a cut in the process execution cycle and inject a condition for it to resume to Umode. The cut point is labeled X, just before the process executes goUmode(). In order to create a suitable condition for a process to goUmode(), we may ask the question: how did a process come to the cut point? The sequence of events must be as follows:

- (1). It did an INT 80 in Umode by  

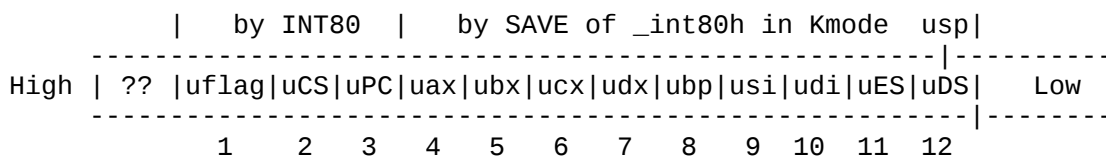
INT 80 -----> \_int80h: in Kmode

returnHERE: ret

When the process first enters Kmode, its ustack contains

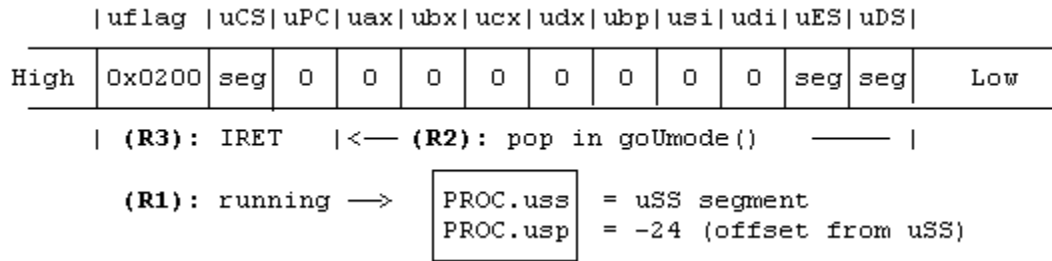


where the saved uPC points to returnHERE: in the Umode code section. Then it executes \_int80h: to save CPU registers to its ustack, which becomes



Then it saves the Umode segment in its PROC.uss and the stack pointer in PROC.usp. Our task here is to create a ustack for the newly created process to goUmode(). We shall fill in the ustack contents as if it was done by the process itself when it entered Kmode. After loading the Umode image to a segment, the ustack is at the high end of the segment. Although the process never existed before, we pretend that

The process executed INT 80 from the virtual address 0 in Umode, and that is where it shall return to when it goUmode(). Furthermore, before executing INT 80 its ustack was empty, the CPU segment registers CS, DS, ES, SS all pointed at its Umode segment and all other registers were 0's, except uflag, which should allow interrupts while in Umode. Therefore, the saved process context should be as shown in Figure 5.6.



**Figure 5.6. Process Context before goUmode()**

In Figure 5.6, uflag = 0x0200 (I-bit=1 to allow interrupts), uCS = uDs = uES = Umode segment, uPC = 0 (for virtual address 0) and all other "saved" registers are 0. In the process PROC, we set PROC.(uss, usp) = (Umode segment, -24). The reader may wonder why -24? In the ustack, the saved uDS is the 12th entry from the left. Its offset is -2\*12= -24 from the high end of ustack. In 16-bit binary, -24 is 111111111101000 (in 2's-complement form) or 0xFFE8 in hex, which is exactly the offset address of the saved uDS in ustack. In other words, a virtual address in a segment can be expressed either as a positive offset from the low end or as a negative offset from the high end (64KB=0) of the segment.

## 5.7. Execution of User Mode Image

With the ustack contents as shown in Figure 5.6, when the PROC becomes running and executes `_goUmode`, it would do the steps (R1) to (R3) shown in Figure 5.6.

- (R1): Restore CPU's (SS, SP) registers from (PROC.uss, PROC.usp). The stack is now the process ustack as shown in the figure.
- (R2). Pop "saved" registers into CPU, which sets DS and ES to Umode segment.
- (R3). Execute IRET, which pops the remaining 3 items off ustack into CPU's flag, CS, PC registers, causing the CPU to execute from (CS, PC)=(seg, 0), which is the beginning of the Umode image program, i.e. the first instruction `start: call _main in ts.s`.

When execution of the Umode image begins, the ustack is logically empty. As soon as execution starts, the ustack contents will change. As the process continues to execute, the ustack frames will grow and shrink as described in Chapter 2. Note that when control first enters the Umode image, the CPU's bp register is initially 0. Recall that bp is the stack frame pointer in function calls. This is why the Umode stack frame link list ends with a 0.

In the `body()` function, add a 'u' command, which calls `goUmode()` to let the running process return to Umode. Alternatively, we may modify `kfork()` by setting the resume point of every newly created process to `goUmode()`. In that case, a process would return to Umode immediately when it begins to run. The following lists the changes to the MTX5.0 system code.

## 5.8. MTX5.0: Demonstration of Simple System Calls

```

! ----- u.s file -----
        .globl _main, _syscall, _exit, _getcs
        call _main
! if main() returns, call exit(0)
        push #0
        call _exit
_syscall: int    80
        ret
_getcs:  mov    ax, cs
        ret

/***** ucode.c file: syscall interface fucntions *****/
int getpid()      { return syscall(0,0,0,0); }
int ps()          { return syscall(1,0,0,0); }
int chname(char *s) { return syscall(2,s,0,0); }
int kfork()       { return syscall(3,0,0,0); }
int kswitch()     { return syscall(4,0,0,0); }
int wait(int *status) { return syscall(5,status,0,0); }
int exit(int exitValue) { syscall(6,exitValue,0,0); }
/***** common code of Umode programs *****/
char *cmd[]={"getpid","ps","chname","kfork","switch","wait","exit",0};
int show_menu()
{
    printf("***** Menu *****\n");
    printf("* ps chname kfork switch wait exit *\n");
    printf("*****\n");
}
int find_cmd(char *name) // convert cmd to an index
{
    int i=0; char *p=cmd[0];
    while (p){
        if (!strcmp(p, name))
            return i;
        i++;
        p = cmd[i];
    }
    return -1;
}
/***** Umode command functions *****/
int ukfork() { kfork(); }
int uswitch(){ kswitch(); }
int uchname()
{
    char s[32];
    printf("input new name : ");
    chname(gets(s)); // assume gets() return pointer to string
}
int uwait()
{
    int child, status;
    child = wait(&status);
    printf("proc %d, dead child=%d\n", getpid(), child);
    if (child >= 0) // only if has child
        printf("status=%d\n", status);
}
int uexit()
{
    char s[16]; int exitValue
    printf("enter exitValue : ");
    exitValue = atoi(gets(s));
}

```

```

        exit(extiValue);
    }
/***** u1.c file *****/
#include "ucode.c"
main()
{
    char command[64];
    int pid, cmd, segment;
    while(1){
        pid = getpid();           // syscall to get process pid
        segment = getcs();        // getcs() return CS register
        printf("-----\n");
        printf("I am proc %d in U mode: segment=%x\n", pid, segment);
        show_menu();
        printf("Command ? "); gets(command);
        if (command[0]==0) continue;
        cmd = find_cmd(command);
        switch(cmd){
            case 0 : getpid();      break;
            case 1 : ps();          break;
            case 2 : ucname();      break;
            case 3 : ukfork();      break;
            case 4 : uswitch();     break;
            case 5 : uwait();       break;
            case 6 : uexit();       break;
            default: printf("Invalid command %s\n", command); break;
        }
    }
}

```

When a process executes u1 in Umode, it runs in a while(1) loop. After printing its pid and segment, it displays a menu

```

***** Menu *****
* ps  cname  kfork  switch  wait  exit *
*****

```

Then it prompts for an input command. Each command invokes a user command function, which calls a syscall interface function to issue a syscall to the MTX5.0 kernel. Each syscall is routed to a corresponding kernel function by the syscall handler kcinth() in kernel, which is shown below.

```

/***** MTX5.0 syscall handler: ustack layout in syscall *****/
usp 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
--|-----
|uds|ues|udi|usi|ubp|udx|ucx|ubx|uax|upc|ucs|uflag|rPC| a | b | c | d |
-----*/
#define AX 8
#define PA 13
int kcinth()
{
    u16 segment, offset; int a, b, c, d, r;
    segment = running->uss; offset = running->usp;
    /* get syscall parameters from ustack */
    a = get_word(segment, offset + 2*PA);

```

```

b = get_word(segment, offset + 2*(PA+1);
c = get_word(segment, offset + 2*(PA+2);
d = get_word(segment, offset + 2*(PA+3));
/* route syscall call to kernel functions by call# a */
switch(a){
    case 0 : r = getpid();    break;
    case 1 : r = kps();      break;
    case 2 : r = kchname(b); break;
    case 3 : r = kkfork();   break;
    case 4 : r = kswitch();  break;
    case 5 : r = kwait(b);   break;
    case 6 :    kexit(b);    break;
    default: printf("invalid syscall %d\n", a);
}
put_word(r, segment, offset + 2*AX); // return value in uax
}

```

If the number of syscalls is small, it suffices to use a switch table to route the syscalls. When the number of syscalls is large, it is better to use a branch table containing kernel function pointers. The following illustrates the function pointer table technique.

(1). Define kernel function prototypes.

```
int kgetpid(), kps(), kchname(), kkfork(), kswitch(), kwait(), kexit();
```

(2). Set up a table of function pointers, each index corresponds to a syscall number.

```

          0          1          2          3          4          5          6
int (*f[ 7 ])()={kgetpid, kps, kchname, kkfork, kswitch, kwait, kexit};

```

(3). Call the kernel function by the call# a, passing parameters to the function as needed.

```
int r = (*f[a])(b,c,d);
```

At this point, our purpose is to show the control flow of syscalls. Exactly what each syscall does is unimportant. The user commands are designed in such a way that the corresponding kernel functions either already exist or are very easy to implement. For example, kgetpid() returns the running process pid, kps() prints the status information of the PROCs, kkfork() calls kfork() to create a child process and returns its pid, kswitch() calls tswitch() to switch process, kwait() and kexit() already exist in the MTX kernel. The chname syscall is intended to show the different address spaces of Umode and Kmode. Each PROC has a char name[32] name field, which is initialized with the name of a heavenly body in the Solar system. The chname syscall changes the running PROC's name to a new string. Before fully understand the distinction between user and kernel mode address spaces, many students try to implement the change name function in kernel as

```

int kchname(char *newname)
{
    strcpy(running->name, newname); return 0 for SUCCESS;
}

```

To their dismay, it only changes the PROC's name to some unrecognizable garbage or, even worse, causes their kernel to crash. The reader is encouraged to figure out the reason and try to implement it correctly.

### 5.8.1. Validation of System Call Parameters

Besides illustrating the difference between user and kernel mode address spaces, the `chname` syscall is also intended to bring out another important point, namely the kernel must validate system call parameters. In the `chname(char *newname)` syscall, the parameter `newname` is a virtual address in user space. What if it is an invalid address? In the real mode MTX, this can not happen since an offset in a segment is always a valid virtual address. In systems with memory protection hardware, the kernel may generate an exception if it tries to access an invalid address. In addition, the `chname` syscall also has an implicit value parameter, the length of the `newname` string. In the MTX5.0 kernel, the name field of each PROC only has room for 32 chars. What if the user tries to pass in a `newname` string longer than 32 chars? If the kernel simply accepts the entire string, it may overflow the PROC's name field and write to some other areas in kernel space, causing the kernel to crash. Similarly, in some syscalls the kernel may write information to user space. If the syscall address parameter is invalid, the kernel may generate a protection error or write to the wrong process image. For these reasons, the kernel must validate all syscall parameters before processing the syscall.

MTX5.0 demonstrates Umode to Kmode transitions and simple syscalls. Figure 5.7 shows the sample outputs of running MTX5.0 under QEMU. The reader may run the system to test other syscall commands.

```
proc 1 running: parent = 0  enter a char [sifiwiqiul] : u

I am proc 1 in U mode: running segment=0x2000
===== Menu =====
* ps chname kfork switch wait exit *
=====
Command ? ps
=====
  name      status    pid    ppid
-----
Sun         READY      0      0
Mercury     running    1      0
Venus       FREE
Earth       FREE
Mars        FREE
Jupiter     FREE
Saturn      FREE
Uranus      FREE
Neptune     FREE

I am proc 1 in U mode: running segment=0x2000
===== Menu =====
* ps chname kfork switch wait exit *
=====
Command ?
```

Figure 5.7. Sample Outputs of MTX5.0

## 5.9. fork-exec in Unix/Linux

In Unix/Linux, the system call  
`int pid = fork();`



creates a child process with a Umode image identical to that of the parent. If successful, `fork()` returns the child process pid. Otherwise, it returns -1. When the child process runs, it returns to its own Umode image and the returned pid is 0. This is the basis of the C code in user mode programs

```
int pid = fork(); // fork a child process
if (pid){ // parent executes this part; }
else { // child executes this part; }
```

The code uses the returned pid to differentiate between the parent and child processes. Upon return from `fork()`, the child process usually uses the system call

```
int r = exec(char *filename);
```

to change image to that of a different program and return to Umode to execute the new image. If successful, `exec()` merely replaces the original Umode image with a new image. It is still the same process but with a different Umode image. This allows a process to execute different programs. In general, `exec` takes more parameters than a single filename. The extra parameters, known as command line parameters, are passed to the new image when execution starts. To begin with, we shall consider `exec` with only a filename first, and consider command line parameters later.

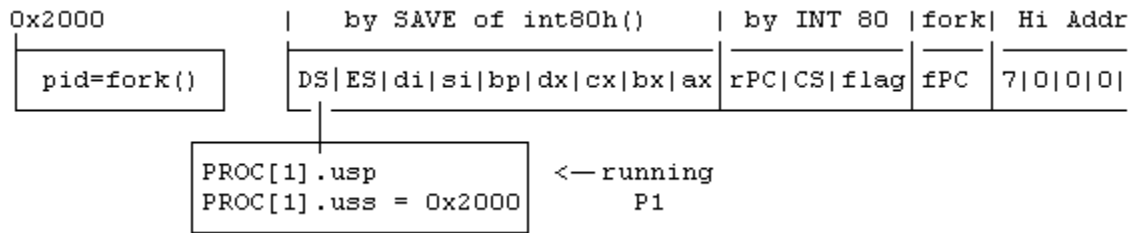
Fork and `exec` may be called the bread and butter of Unix because almost every operation in Unix depends on fork-exec. For example, when a user enters a command, the `sh` process forks a child process and waits for the child to terminate. The child process uses `exec` to change its image to the command file and executes the command. When the child process terminates, it wakes up the parent `sh`, which prompts for another command, etc. While Unix uses the fork-exec paradigm, which creates a process to execute a different program in two steps, there are alternative schemes. In the MVS [IBM MVS] operating system, the system call `create(filename)` creates a child process to execute filename in one step, and `attach(filename)` allows a process to execute a new file without destroying the original execution image. Variations to the fork-exec paradigm are listed in the Problem section as programming exercises. In the following, we shall implement fork and `exec` exactly the same as they are in Unix.

### 5.9.1. Implementation of fork in MTX

First, we outline the algorithm of fork. Then we explain the steps by a specific example.

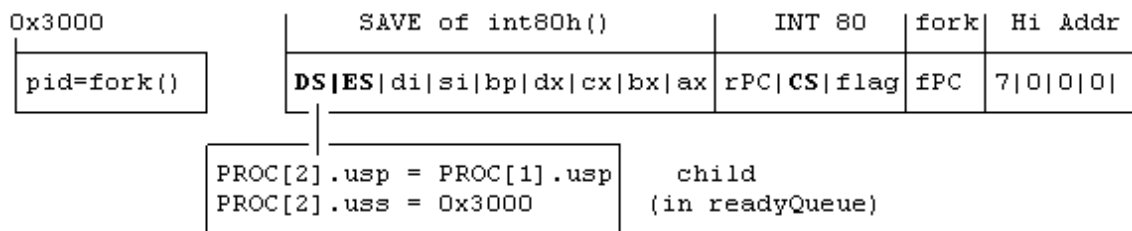
```
/****** Algorithm of fork *****/
1. create a child proc ready to run in Kmode, return -1 if fails;
2. copy parent Umode image to child Umode segment;
3. set child PROC.(uss, usp)=(child segment, parent PROC.usp);
4. fix up child's ustack to let it return to Umode image with 0;
5. return child pid
```

Assume that process P1 calls `pid=fork()` from Umode, as shown in Figure 5.8.



**Figure 5.8. fork() Diagram of Parent Process**

In Figure 5.8, the left-hand shows P1's Code and Data sections and the right-hand side shows changes to its ustack. When P1 executes fork(), it calls syscall(7,0,0,0) in assembly. The saved fPC on its ustack points to the return point in fork(). In \_syscall, it executes INT 80 to enter the MTX kernel. In int80h(), it saves CPU registers in ustack and saves uSS = 0x2000 and uSP into PROC[1].(uss, usp). Then, it calls kfork(0) to create a child process P2 in the segment 0x3000 but does not load any Umode image file. Instead, it copies the entire Umode image of P1 to the child segment. This makes the child's Umode image identical to that of the parent. Figure 5.9 shows the copied image of the child process.



**Figure 5.9. Copied User Mode Image of Child Process**

Since PROC[2]'s Umode image is in the segment 0x3000, its saved uss must be set to 0x3000. Since usp is an offset relative to the PROC's segment, PROC[2]'s saved usp should be the same as that of the parent. However, if we let the child return to Umode as is, it would goUmode with the ustack contents as shown in Figure 5.9, causing it to return to the segment 0x2000 since the copied DS, ES, CS in its ustack are all 0x2000. This would send P2 to execute in P1's segment also, like sending two fellows to the same bed. Interesting perhaps but not very good since the processes will interfere with each other. What we need is to send P2 back to its own bed (segment). So we must fix up P2's ustack before letting it goUmode. In order to let the child process return to its own segment, we must change the copied DS, ES and CS to child's segment 0x3000. All other entries in the copied image, such as rPC and fPC, do not need any change since they are offset values relative to a segment. To let the child return pid=0, simply change its saved uax to 0. With these modifications, the child will return to a Umode image identical to that of the parent but in its own segment 0x3000. Because of the copied rPC and fPC values, the child will return to the same place as does the parent, i.e. to the statement pid = fork(); as if it had called fork() before, except that the returned pid is 0. The C code of fork() in the MTX5.1 kernel is shown below.

```

/***** fork() in MTX5.1 kernel *****/

```

```

int copyImage(u16 pseg, u16 cseg, u16 size)
{
    u16 i;
    for (i=0; i<size; i++)
        put_word(get_word(pseg, 2*i), cseg, 2*i);
}

int fork()
{
    int pid; u16 segment;
    PROC *p = kfork(0);          // kfork() a child, do not load image file
    if (p==0) return -1;          // kfork failed
    segment = (p->pid+1)*0x1000; // child segment
    copyImage(running->uss, segment, 32*1024); // copy 32K words
    p->uss = segment;             // child's own segment
    p->usp = running->usp;         // same as parent's usp
    /*** change uDS, uES, uCS, AX in child's ustack ***/
    put_word(segment, segment, p->usp);          // uDS=segment
    put_word(segment, segment, p->usp+2);        // uES=segment
    put_word(0, segment, p->usp+2*8);            // uax=0
    put_word(segment, segment, p->usp+2*10);     // uCS=segment
    return p->pid;
}

```

### 5.9.2. Implementation of exec in MTX

The implementation of exec is also very simple. The algorithm of exec is

```

/***** Algorithm of exec(filename) *****/
1. get filename from Umode space;
2. load filename to running proc's segment; return -1 if fails;
3. initialize proc's ustack for it to execute from VA=0 in Umode.

```

Note that exec() return -1 if the operation fails. It does not return any value on success. In fact, it never returns. The reader is encouraged to figure out why? The C code of kexec() is shown below.

```

/***** kexec(filename) in MTX5.1 kernel *****/
int kexec(char *y) // y points at filename in Umode space
{
    int i, length = 0;
    char filename[64], *cp = filename;
    u16 segment = running->uss; // same segment
    /* get filename from U space with a length limit of 64 */
    while( (*cp++ = get_byte(running->uss, y++)) && length++ < 64 );
    if (!load(filename, segment)); // load filename to segment
        return -1; // if load failed, return -1 to Umode
    /* re-initialize process ustack for it return to VA=0 */
    for (i=1; i<=12; i++)
        put_word(0, segment, -2*i);
    running->usp = -24; // new usp = -24
    /* -1 -2 -3 -4 -5 -6 -7 -8 -9 -10 -11 -12 ustack layout */
    /* flag uCS uPC ax bx cx dx bp si di uES uDS */
    put_word(segment, segment, -2*12); // saved uDS=segment
    put_word(segment, segment, -2*11); // saved uES=segment
    put_word(segment, segment, -2*2); // uCS=segment; uPC=0
    put_word(0x0200, segment, -2*1); // Umode flag=0x0200
}

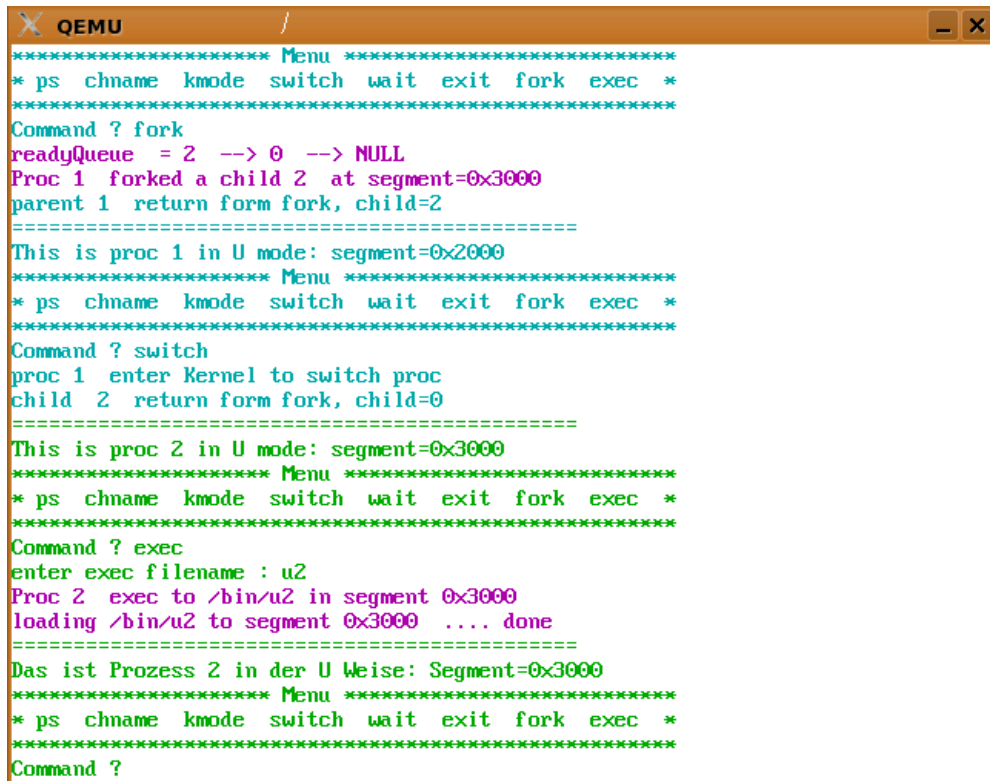
```

### 5.9.3. MTX5.1: Demonstration of fork-exec in MTX

To test fork and exec, we add the user commands fork, exec and the corresponding syscall interface functions to MTX5.1.

```
int fork()      { return syscall(7,0,0,0); }
int exec(char *s){ return syscall(8,s,0,0); }
int ufork()     // user fork command
{   int child = fork();
    (child)? printf("parent ") : printf("child ")
    printf("%d return form fork, child_pid=%d\n", getpid(), child);
}
int uexec()     // user exec command
{   int r; char filename[64];
    printf("enter exec filename : ");
    gets(filename);
    r = exec(filename);
    printf("exec failed\n");
}
```

In order to demonstrate exec, we need a different Umode image. The image file u2 is the same as u1 except that it displays in German. The reader may create other Umode image files to speak different languages, just for fun. Figure 5.10 shows the sample outputs of running MTX5.1.



```
***** Menu *****
* ps cname kmode switch wait exit fork exec *
*****
Command ? fork
readyQueue = 2 --> 0 --> NULL
Proc 1 forked a child 2 at segment=0x3000
parent 1 return form fork, child=2
=====
This is proc 1 in U mode: segment=0x2000
***** Menu *****
* ps cname kmode switch wait exit fork exec *
*****
Command ? switch
proc 1 enter Kernel to switch proc
child 2 return form fork, child=0
=====
This is proc 2 in U mode: segment=0x3000
***** Menu *****
* ps cname kmode switch wait exit fork exec *
*****
Command ? exec
enter exec filename : u2
Proc 2 exec to /bin/u2 in segment 0x3000
loading /bin/u2 to segment 0x3000 .... done
=====
Das ist Prozess 2 in der U Weise: Segment=0x3000
***** Menu *****
* ps cname kmode switch wait exit fork exec *
*****
Command ?
```

Figure 5.10. Sample Outputs of MTX5.1

## 5.10. Command line parameters

In Unix/Linux, when a user enter a command line = “cmd a1 a2 ... an”, sh forks a child process to execute the cmd program, which can be written as

```
main(int argc, char *argv[], char *env[ ])
{
    // argc, argv are passed to main() by exec()
}
```

Upon entry to main(), argc = n+1, argv points to a null terminated array of string pointers, each points to a parameter string, as shown in Figure 5.11.

```
argv -> [ . | . | ..... | . | 0 ]
          |   |   .....   |
          "cmd" "a1"         "an"
```

**Figure 5.11. Command-Line Parameters**

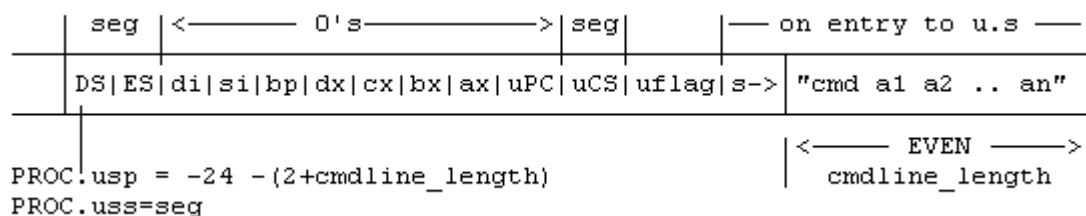
The env parameter points to a null terminated array containing environment string pointers similar to argv. By convention, argv[0] is the program name and argv[1] to argv[argc-1] are command line parameters to the program. In Unix/Linux, the command line parameters are assembled in Umode before calling kexec() in kernel via the syscall

```
execve(char *cmd, char *argv[ ], char *env[ ]);
```

The Unix kernel passes argv and env to the new image. Because of the limited space in the MTX kernel, we shall implement command line parameters in a different way, but with the same end results. In MTX, the exec syscall takes the entire command line as parameter, i.e.

```
int r = exec("cmd a1 a2 ... an");
```

We modify kexec() in kernel by using the first token, cmd, as the filename. After loading the filename, we set up the ustack of the new Umode image as shown in Figure 5.12.



**Figure 5.12. Command Line Parameter in User Mode Stack**

First, we pad the command line with an extra byte to make the total length even, if necessary. Next, we put the entire string into the high end of ustack and let s point at the string in ustack. Then we create a syscall interrupt stack frame for the process to goUmode, as shown in the left-hand side of Figure 5.12. When execution begins in Umode, the ustack top contains s, which points to the command string in ustack. Accordingly, we modify u.s to call main0(), which parses the command line into string tokens and then calls main(int argc, char \*argv[]). The algorithm of main0() is shown below.

```

main0(char *s) // *s is the original command line "cmd a1 a2 ... an"
{
    tokenize *s into char *argv[ ] with argc = number of token strings;
    main(argc, argv);
}

```

## 5.11. Simple sh for Command Execution

With fork and exec, we can standardize the execution of user commands by a simple sh. First, we precompile main0.c as crt0.o and put it into the link library mtllib as the C startup code of all MTX Umode programs. Then we write Umode programs in C as

```

/***** filename.c file *****/
#include "ucode.c" // user commands and syscall interface
main(int argc, char *argv[ ])
{ // C code of Umode program }

```

Then we implement a rudimentary sh for command execution as follows.

```

/***** sh.c file *****/
#include "ucode.c" // user commands and syscall interface
main(int argc, char *argv[ ])
{
    int pid, status;
    while(1){
        display executable commands in /bin directory
        prompt for a command line cmdline = "cmd a1 a2 .... an"
        if (!strcmp(cmd,"exit"))
            exit(0);
        // fork a child process to execute the cmd line
        pid = fork();
        if (pid) // parent sh waits for child to die
            pid = wait(&status);
        else // child exec cmdline
            exec(cmdline); // exec("cmd a1 a2 ... an");
    }
}

```

Then compile all Umode programs as binary executables in the /bin directory and run sh when the MTX starts. This can be improved further by changing P1's Umode image to an init.c file.

```

/***** init.c file : initial Umode image of P1 *****/
main( )
{
    int sh, pid, status;
    sh = fork();
    if (sh){ // P1 runs in a while(1) loop
        while(1){
            pid = wait(&status); // wait for ANY child to die
            if (pid==sh){ // if sh died, fork another one
                sh = fork();
                continue;
            }
        }
    }
}

```

```

        printf("P1: I just buried an orphan %d\n", pid);
    }
}
else
    exec("sh");          // child of P1 runs sh
}

```

MTX5.2 demonstrates a simple sh for command execution. In MTX5.2, P1 is the INIT process, which executes the /bin/init file. It forks a child process P2 and waits for any ZOMBIE children. P2 exec to /bin/sh to become the sh process. The sh process runs in a while(1) loop, in which it displays a menu and asks for a command line to execute. All commands in the menu are user mode programs. When the user enters a command line of the form

cmd parameter-list

sh forks a child process to execute the command line and waits for the child process to terminate. The child process uses exec to execute the cmd file, passing parameter-list to the program. When the child process terminates, it wake up the sh process, which prompts for another command line. When the sh process itself terminates, the INIT process P1 forks another sh process, etc.

Eventually, we shall expand init.c to let P1 fork several login processes on different terminals for users to login. When a user login, the login process becomes the user process and executes sh. This would make the running environment of MTX identical to that of Unix/Linux. Figure 5.13 shows the sample outputs of MTX5.2

```

QEMU
Proc 1 forked a child 2 at segment=0x3000
init waits
Proc 2 exec to /bin/sh in segment 0x3000
loading /bin/sh to segment 0x3000 .... done
sh 2 running
===== commands in /bin =====
newname ps fork exec u1 u2
=====
enter command (cmd in /bin OR exit) : exec u1 test command line parameters
Parent sh 2 fork a child
readyQueue = 3 --> 0 --> NULL
Proc 2 forked a child 3 at segment=0x4000
parent sh 2 waits
child sh 3 running
Proc 3 exec to /bin/exec in segment 0x4000
loading /bin/exec to segment 0x4000 .... done
Proc 3 exec to /bin/u1 in segment 0x4000
loading /bin/u1 to segment 0x4000 .... done
enter main() : argc = 5
argv[0] = u1
argv[1] = test
argv[2] = command
argv[3] = line
argv[4] = parameters
=====
I am proc 3 in U mode: segment=0x4000
***** Menu *****
* ps cname kmode switch wait exit fork exec *
*****
Command ?

```

Figure 5. 13. Sample Outputs of MTX5.2

## 5.12. vfork

In a Unix-like system the usual behaviors of parent and child processes are as follows.

```
if (fork())           // parent fork() a child process
    wait(&status);    // parent waits for child to terminate
else
    exec(filename);    // child executes a new image file
```

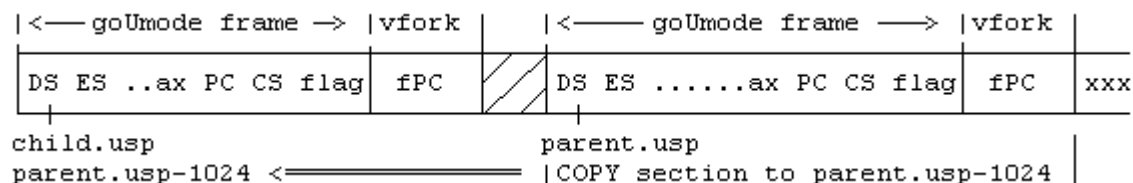
After creating a child, the parent waits for the child to terminate. When the child runs, it changes Umode image to a new file. In this case, copying image in fork() would be a waste since the child process abandons the copied image immediately. For this reason, most Unix systems support a vfork, which is similar to fork but does not copy the parent image. Instead, the child process is created to share the same image with the parent. When the child does exec, it only detaches itself from the shared image without destroying it. If every child process behaves this way, the scheme would work fine. But what if users do not obey this rule and allow the child to modify the shared image? It would alter the shared image, causing problems to both processes. To prevent this, the system must rely on memory protection. In systems with memory protection hardware, the shared image can be marked as read-only so that a process can only execute the image but not modify it. If either process tries to modify the shared image, the image must be split into separate images. So far, MTX runs on Intel x86 machines in real mode, which does not have memory protection mechanism. Despite this, we can implement vfork also, provided that the child process only does exec without modifying the shared image.

### 5.12.1. Implementation of vfork in MTX

The algorithm of vfork is as follows.

```
/****** Algorithm of vfork *****/
1. kfork(0) a child process ready to run in Kmode, return -1 if fails;
2. copy a section of parent's ustack from parent.usp all the way back
   to where it called pid = vfork(), as shown in Figure 5.14.
3. Let child PROC.uss = parent PROC.uss, so that they share the same
   segment, set child PROC.usp = parent PROC.usp-1024, change ax in
   child's goUmode frame to 0 for it to return 0 to pid=vfork();
4. return child pid;
```

Figure 5.14 illustrates Step 2 of the vfork algorithm.



**Figure 5.14. goUmode Stack Frames in vfork()**



In Figure 5.14, the right-hand side shows the Umode stack frame of the parent process in a `vfork()` system call. The stack frame is copied to a low area in the parent's ustack as the Umode stack frame of the vforked child process, as shown on the left-hand side of the figure. When the parent returns, it executes `goUmode()` by the stack frame from `parent.usp`, and the returned value is `child pid`. When the child runs, it executes `goUmode()` by the stack frame from `child.usp`. The stack frames are identical, except for the return values. Therefore, the child also returns to `pid = vfork()` with a 0. When the child returns to Umode, it runs in the same image as the parent. Then it issues an `exec` syscall to change image. When it enters `kexec()`, the Umode segment is still that of the parent. The child can fetch the command line and then `exec` to its own segment. To support `vfork`, `kexec()` only needs a slight modification. If the caller is a vforked process, instead of the caller's current segment, it loads the new image to the caller's default segment (by `pid`), thereby detaching it from the parent image. The following shows the `vfork` code in the MTX5.3 kernel.

```

/***** vfork() in MTX5.3 kernel *****/
int vfork()
{
    int pid, i, w;  u16 segment;
    PROC *p = kfork(0);    // kfork() child, do not load image file
    if (p==0)
        return -1;        // kfork() failed
    p->vforked = 1;        // set vforked flag, used in kexec()
    /* copy a section of parent ustack for child to return to Umode */
    for (i=0; i<24; i++){ // 24 words is enough; > 24 should also work
        w = get_word(running->uss, running->usp+i*2);
        put_word(w, running->uss, running->usp-1024+i*2);
    }
    p->uss = running->uss;
    p->usp = running->usp - 1024;
    put_word(0, running->uss, p->usp+8*2); // set child's return value to 0
    p->kstack[SSIZE-1] = goUmode;        // child goUmode directly
    return p->pid;
}

```

### 5.12.2. MTX5.3: Demonstration of `vfork` in MTX

MTX5.3 demonstrates `vfork` in MTX. To do this, we add a `uvfork` syscall interface and a user mode `vfork` command program.

```

/***** User mode Program of vfork() *****/
int uvfork(){return syscall(9,0,0,0);} // vfork() system call
int vfork(int argc, char *argv[ ])
{
    int pid, status;
    pid = uvfork(); // call uvfork() to vfork a child
    if (pid){
        printf("vfork parent %d waits\n", getpid());
        pid = wait(&status);
        printf("parent %d waited, dead child=%d\n", getpid(), pid);
    }
    else{ // vforked child process

```

```

    printf("vforked child %d in segment %x\n", getpid(), getcs());
    printf("This is Goldilocks playing in Papa bear's bed\n");
    printf("EXEC NOW! before he wakes up\n");
    exec("u2 Bye Bye! Papa Bear");
    printf("exec failed! Goldilocks in deep trouble\n");
    exit(1); // better die with dignity than mauled by a big bear
}
}

```

When running the vfork command it will show that every vforked child process begins execution in the same segment of the parent, like Goldilocks playing in Papa bear's bed. If the child process exec to its own segment before the parent runs or while the parent is sleeping in wait(), the Papa bear would never know Goldilocks played in his bed before. Figure 5.15 shows the sample outputs of running the vfork command in MTX5.3.

```

This is proc 1 in U mode: segment=0x2000
Menu *****
* ps cname kmode switch wait exit fork exec vfork *
*****
Command ? vfork
vfork() in kernel Proc 1 forked a child 2 at segment=0x3000
fix ustack for child to return to Umode
vfork parent 1 waits
proc 1 enter Kernel to wait for a child to die
vforked child 2 in segment=0x2000
This is Goldilocks playing in Papa bear's bed
EXEC NOW! before he wakes up
Proc 2 exec to /bin/u2 in segment 0x3000
loading /bin/u2 to segment 0x3000 .... done
main0: s=u2 Bye Bye! Papa Bear
enter main : argc = 5
argv[0 ] = u2
argv[1 ] = Bye
argv[2 ] = Bye!
argv[3 ] = Papa
argv[4 ] = Bear
=====
Das ist Prozess 2 in der U Weise: Segment=0x3000

```

Figure 5.15. Sample Outputs of MTX5.3

## 5.13. Threads

### 5.13.1. Principle of Threads

Threads are independent execution units in the same address space of a process. In the process model, each process is an independent execution unit but each process has only one execution path. In the thread model, a process contains multiple threads, each of which is an independent execution unit. When creating a process it is created in a unique address space with a main thread. When a process begins to run, it runs the main thread of the process. With only a main thread, there is virtually no difference between a process and a thread. However, the main thread may create other threads. Each thread may create yet more threads, etc. All threads in a process execute in the same address space of the process but each thread is an independent execution unit. In addition to sharing a

common address space, threads also share many other resources of a process, such as user id, opened file descriptors and signals, etc. A simple analogy is that a process is a house with a house master (the main thread). Threads are people living in the same house of a process. Each person in a house can carry on his/her daily activities independently, but they share some common facilities, such as the same mailbox, kitchen and bathroom, etc. Historically, most OS used to support their own proprietary threads. Currently, almost all OS support Pthreads, which is the threads standard of IEEE POSIX 1003.1c [POSIX, 1995]. For more information, the reader may consult numerous books [Buttler et al., 1996] and on-line articles on Pthreads programming [Pthreads]. Threads have many advantages over processes.

### **5.13.2. Advantages of Threads**

#### **(1). Thread creation and switching are faster**

In general, creating a thread is faster than creating a process since it does not need to allocate memory space for the thread image. Also, thread switching is faster than process switching. The context of a process is complex and large. The complexity stems mainly from the process image. For example, in a system with virtual memory, a process image may be composed of many pages. During execution some of the pages are in memory while others are not. The OS kernel must use several page tables and many levels of hardware assistances to keep track of the pages of each process. Process switching involves replacing the complex paging environment of one process with that of another, which requires a lot of operations and time. In contrast, threads of a process share the same address space. Switching among threads in the same process is much simpler and faster because the OS kernel only needs to switch the execution points without changing the process image.

#### **(2). Threads are more responsive**

A process has only a single execution path. When a process becomes blocked, the entire process execution stops. In contrast, when a thread is suspended, other threads in the same process can continue to execute. This allows a program with threads to be more responsive. For example, in a process with threads, while one thread is blocked for I/O, other threads can still do computations in the background. In a server with threads, the server can serve multiple clients concurrently.

#### **(3). Threads are better suited to parallel computing**

The goal of parallel computing is to use multiple execution paths to solve problems faster. Algorithms based on the principle of divide and conquer, e.g. binary search and quicksort, etc. often exhibit a high degree of parallelism, which can be exploited by using parallel or concurrent executions to speed up the computation. Such algorithms often require the executions to share common data. In the process model, processes cannot share data efficiently because their address spaces are all distinct. To remedy this, processes must use Inter-Process Communication (IPC) or other means to include a common data area in their address spaces. In contrast, threads in the same process share all the (global) data in the same address space. Thus, writing programs for parallel executions using threads is simpler and more natural than using processes.

### 5.13.3. Threads Operations

The execution locus of a thread is similar to that a process, i.e. it can execute in either Umode or Kmode. In Umode, threads run in the same address space of a process but each has its own execution stack. As an independent execution unit, a thread can make syscalls to the OS kernel, subject to the kernel's scheduling policy, becomes suspended, and resumes to continue execution, etc. From the execution locus point of view, threads are just like processes. To take advantage of the share address space of threads, the kernel's scheduling policy may favor threads of the same process over those in different processes. As of now, almost all OS provide Pthreads compatible threads support, which includes

- .thread creation and termination: `thread_create()`, `thread_exit()`, etc.
- .threads synchronization: `thread_join()`, `mutex`, `condition_variables`, etc.

In the following, we shall discuss the implementation of threads in MTX.

## 5.14. Implementation of Threads in MTX

Ideally, a PROC structure should contain a process part and a thread part. The process part contains process information that is common to all threads in the process. The thread part contains information that is unique to each thread. Rather than drastically altering the PROC structure, we shall add a few new fields to the current PROC structure and use it for both processes and threads.

### 5.14.1. Thread PROC structure

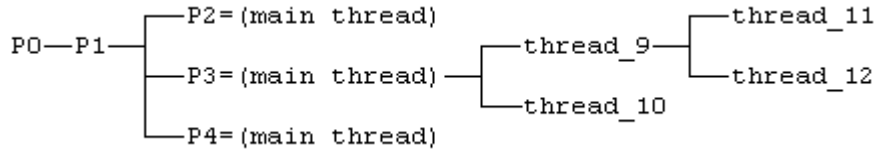
```
typedef struct proc{
    /* same as current PROC sturct but add new entries */
    int type;                // PROCESS|THREAD
    int tcount               // total number of threads in PROC
    struct proc *tlist;      // a list of all threads in PROC
    struct resouce *resourcePtr; // pointer to process "resource"
    int kstack[SSIZE]        // kstack[] must be the last entry
} PROC;
```

Add NTHREAD PROCs to the MTX kernel, as in

```
#define NPROC    9           // same as before
#define NTHREAD 16          // added PROCs for threads
PROC proc[NPROC+NTHREAD];
```

The first NPROC PROCs are for processes and the remaining ones are for threads. Free process PROCs are maintained in a freeList as before. Free thread PROCs are maintained in a separate tfreeList. When creating a process by fork/vfork we allocate a process PROC from freeList and the PROC type is PROCESS. When creating a thread we allocate a thread PROC from tfreeList and the PROC type is THREAD. When a process is created by fork/vfork, the process PROC is also the main thread of the new process. When creating a thread the new PROC is a thread in the same process of the caller. When

a thread creates another thread, the usual parent-child relation applies. In MTX, P0 runs only in Kmode. All processes with Umode images, begin from P1. Thus, user processes form a process family-tree with P1 as the root. Similarly, threads in a process form a thread family-tree with the main thread as the root, as shown in Figure 5.16.



**Figure 5.16. Process and Thread Family Tree**

In order to maintain the parent-child relation of processes, we require that P1 cannot die if there are other processes still existing. Similarly, the main thread of a process cannot die if there are other threads still existing in the process. From a thread PROC, we can always find the main thread PROC by following the parent PROC pointers. Therefore, there is no need to actually implement the thread family-tree in a process. In the main thread PROC, tcount records the total number of threads in the process. For thread PROCs, tcount is not used. In every PROC, the resourcePtr points to a resource structure containing the per-process resources, such as uid, opened file descriptors and signals, etc. The per-process resource is allocated once only in the main thread. All other threads in a process point to the same resource. At this point, MTX processes do not have any resource yet, so the resource pointer field is not used until later. The added thread PROCs allow for an easy extension of MTX to support threads. For instance, the system falls back to the pure process model if NTHREAD = 0.

### 5.14.2. Threads Management Functions

Threads management in MTX includes the following functions

```

thread creation      : thread(int *fn, *stack, flag, *ptr);
thread termination   : texit(int exitValue);
thread join          : tjoin(int n);
thread synchronization : mutex_create(), mutex_destroy(),
                        mutex_lock(), mutex_unlock();
  
```

### 5.14.3 Thread Creation

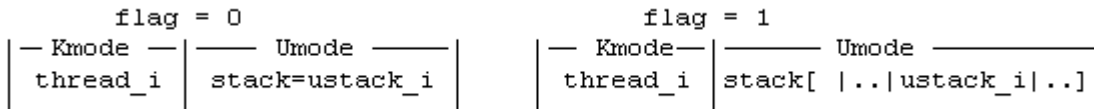
Thread creation in MTX is similar to Linux's clone(). The thread creation function is

```

int kthread(int fn, int *stack, int flag, int *ptr)
{
    // create a thread, where
    fn    = function entry address to be executed by the thread,
    stack = high end address of a ustack area,
    flag  = thread options; 0 or 1 to determine thread's ustack
    ptr   = pointer to a list of parameters for fn(ptr);
}
  
```

The function kthread() creates a thread to execute the function fn(ptr) with a ustack at the high end of the stack area. In Linux's clone() syscall, the flag parameter allows the thread

to have many options, e.g. keep newly opened file descriptors private, etc. For simplicity reasons, we shall not support such options. Besides, our MTX processes do not have any resource yet anyway. So the flag parameter is only used to determine the thread's ustack. When flag=0, the parameter stack is the high end of the thread's ustack. When flag=1, stack is the high end of a total ustack area. The ustack of each thread is a section of the total ustack area determined by its pid. This simplifies the ustack area management when threads are created and terminated dynamically. The ustack usage is shown in Figure 5.17.



**Figure 5.17. Umode Thread Stack Assignment**

Due to the limited number of thread PROCs and the Umode address space size of each process (64KB), we set TMAX <= NTHREAD=16 as the maximal number of threads in a process. The algorithm of thread creation is as follows.

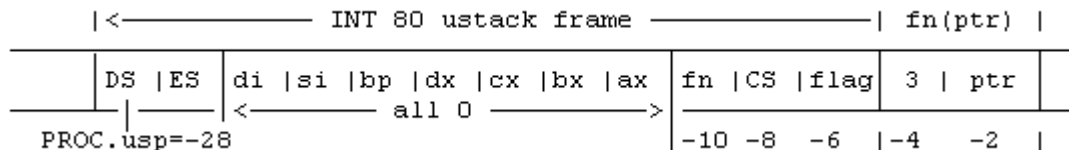
#### Algorithm of thread creation in MTX

- ```

-----
1. if process tcount > TMAX return -1 for error
2. allocate a free THREAD PROC and initialize it ready to run as before,
   i.e. initialize its kstack[ ] to resume to goUmode() directly.
3. determine its ustack high end, initialize the ustack for it return
   to execute fn(ptr) in the Umode image of the caller.
4. enter thread PROC into readyQueue, increment process t_count by 1;
5. return thread pid;
-----

```

Most of the steps are self-explanatory. We only explain step 3 in more detail. In order for a thread to return to Umode to execute fn(ptr), we pretend again that it had done an INT 80 from the virtual address fn. Therefore, its ustack must contain an INT80 stack frame for it to return to Umode. Furthermore, when execution begins from fn, it appears as if the thread had called fn with the parameter ptr. Accordingly, we set up its ustack to contain the contents as shown in Figure 5.18.



**Figure 5.18. Thread Umode Stack Contents**

In Figure 5.18, DS=ES=CS = caller's segment, all "saved registers" are 0's, savedPC = fn and flag=0x0200. When the thread executes goUmode(), it will return to (CS, fn) with the ustack top containing |3|ptr|, as if it had called fn(ptr) from the virtual address 3. At the virtual address 3 (in u.s) is a call to exit(0). When fn() finishes, it returns to the virtual address 3, causing the thread to terminate by exit(0). In addition, we also impose the following restrictions on threads.

- (1). When a process with threads does fork or vfork, it creates a new process with only the main thread, regardless of how many threads are in the caller's process.
- (2). A thread is allowed to exec if and only if the total number of threads in the process is 1. This is clearly necessary because exec changes the entire execution image of a process. If a thread changes the process image, other threads cannot survive. So we modify kexec() in the MTX kernel to enforce this rule, which effectively says that only the main thread with no child threads may change image.

#### 5.14.4. Thread Termination

In the process model, when a process terminates, it sends all orphaned children to P1, which can not die if there are other processes still existing. Similarly, in the thread model, the main thread of a process can not die if there are other threads still existing in the process, for the following reasons. First, if the main thread dies first, the entire process address space would be freed. This would make the "house" of the remaining threads disappear, which is clearly unacceptable. Second, if the main thread dies first, the children threads would have no parent. If the dying main thread sends the orphaned children to P1, P1 cannot maintain the thread count of all the orphaned threads. If any of the surviving thread continues to create threads, P1 would not know what to do with this. So a dying thread should not send its orphans to P1 but to the main thread. This implies that the main thread cannot die if there are other threads in the process. In order for process with threads to terminate properly, we modify kexit() in the MTX kernel as follows, which can be used by both processes and threads to terminate.

```
kexit(int exitValue)
{
    if (caller is a THREAD){
        dec process t_count by 1;
        send all children, if any, to main thread of process;
    }
    else { // caller is a PROCESS)
        while (process t_count > 1)
            wait(&status); // wait for all other threads to die
            /* caller is a process with only the main thread */
            send all children to P1;
        }
        record exitValue in PROC;
        become a ZOMBIE;
        wakeup parent;
        if (has sent children to P1) wakeup P1;
        tswitch();
    }
```

#### 5.14.5. Thread Join Operation

The thread join(thread\_id) operation suspends a thread until the designated thread terminates. In general, a thread may join with any other thread. In practice, a thread usually joins with either a specific child or all of its children. To support the former, we may implement a wait4(chid\_pid) function, which waits for a specific child process to

die. This is left as an exercise. To support join with all children, the current kwait() function is sufficient, as in

```
join(int n) // wait for n children threads to terminate
{
    int pid, status;
    while(n--)
        pid = wait(&status);
}
```

#### 5.14.6. Threads Synchronization

Since threads execute in the same address space of a process, they share all the global variables in a process. When several threads try to modify the same shared variable or data structure, if the outcome depends on the execution order of the threads, it is called a race condition. In concurrent programming, race conditions must not exist. Otherwise, the results may be inconsistent. In order to prevent race conditions among threads, we implement mutex and mutex operations in MTX. A mutex is a data structure

```
struct mutex{
    int status; // FREE|inUSE|LOCKED|UNLOCKED|etc.
    int owner; // owner pid, only owner can unlock
    PROC *queue; // waiting proc queue
}mutex[NMUTEX]; // all initialized as FREE
```

Define the following mutex operations

```
struct mutex *m = mutex_create(); // create a mutex, return pointer
mutex_lock(m) : lock mutex m;
mutex_unlock(m) : unlock mutex
mutex_destroy(m): destroy mutex.
```

Threads use mutex as locks to protect shared data objects. A typical usage of mutex is as follows. A thread first calls `m = mutex_create()` to create a mutex, which is visible to all threads in the same process. A newly created mutex is in the unlocked state and without an owner. Each thread tries to access a shared data object by

```
mutex_lock(m);
    access shared data object;
mutex_unlock(m);
```

When a thread executes `mutex_lock(m)`, if the mutex is unlocked, it locks the mutex, becomes the owner and continues. Otherwise, it blocks and waits in the mutex (FIFO) queue. Only the thread which has acquired the mutex lock can access the shared data object. When the thread finishes with the shared data object, it calls `mutex_unlock(m)` to unlock the mutex. A locked mutex can only be unlocked by the current owner. When unlocking a mutex, if there are no waiting threads in the mutex queue, it unlocks the mutex and the mutex has no owner. Otherwise, it unblocks a waiting thread, which becomes the new owner and the mutex remains locked. When all the threads are finished, the mutex may be destroyed for possible reuse. Although the MTX kernel supports mutex operations, we shall discuss their implementations in Chapter 6 in the context of process synchronization.



#### 5.14.7. Threads Scheduling

Since thread switching is simpler and faster than process switching, the kernel's scheduling policy may favor threads in the same process over threads in different processes. To illustrate this, we implement a user command, `chpri`, which changes the scheduling priority of a process and all the threads in the same process. Correspondingly, we implement `kchpri()` and `reschedule()` functions in the MTX kernel. Whenever a process changes priority, it also calls `resehede()`, which reorders the `readyQueue` by the new priority. If a process has a higher priority than other processes, the kernel will always run a thread from that process until either there are no runnable threads in the process or its priority is changed to a lower value.

### 5.15. Concurrent Program Examples Using Threads

We demonstrate the threads capability of MTX by the following concurrent programs.

**Example 1. Quicksort by threads:** The `qsort.c` file implements quicksort of an array of 10 integers by threads. When the program starts, it runs as the main thread of a MTX process. The main thread calls `qsort(&arg)`, with `arg's` `lowerbound=0` and `upperbound=arraySize-1`. In `qsort()`, the thread picks a pivot element to divide the array range into two parts such that all elements in the left part are less than the pivot and all elements in the right part are greater than the pivot. Then it creates 2 children threads to sort each of the two parts, and waits for the children to die. Each child thread sorts its range by the same algorithm recursively. When all the children threads finish, the main thread resumes. It prints the sorted array and terminates. As is well known, the number of sorting steps of quicksort depends on the order of the unsorted data, which affects the number of threads needed in the `qsort` program. To test `qsort` on larger data arrays, the number of threads, `NTHREAD`, in `type.h` may have to be increased, but keep in mind that each thread `PROC` requires a `kstack` of `SSIZE` bytes and the MTX kernel space in real mode is only 64 KB.

**Example 2. Compute sum of matrix elements by threads:** The `matrix.c` program computes the sum of the elements in a matrix, `int A[8][8]`, by threads. When the program starts, it runs as the main thread of a MTX process. The main thread creates `n<=8` children threads, denoted by `thread[n]`. In the `thread()` call, we set the parameter `flag = 0` to show that the user program must manage the thread stacks. After creating the children threads, the main threads waits for all the children to terminate by `tjoin(n)`. Each child thread `thread[i]` computes the sum of the `i_th` row of the matrix and deposits the partial sum into `sum[i]`. When the main thread resumes, it computes the total sum by adding all the partial sums.

**Example 3. Race condition and threads synchronization:** The `race.c` file is identical to `matrix.c` except that, instead of depositing each row sum into a distinct `sum[i]`, all threads try to update the same total, as in

```
int total = 0;
each thread : compute row_sum;
              read current total;
```

```

        update total by total += row_sum;
=====> utswitch()          // simulate interrupt
        write total back;

```

Before writing the updated total back, we intentionally insert a `utswitch()` syscall to switch process. This creates race conditions among the threads. Each thread writes to the same total with its old value. The final total value depends on the execution order of the threads and the result is most likely incorrect. However, if we protect the updating code by a mutex lock, as in

```

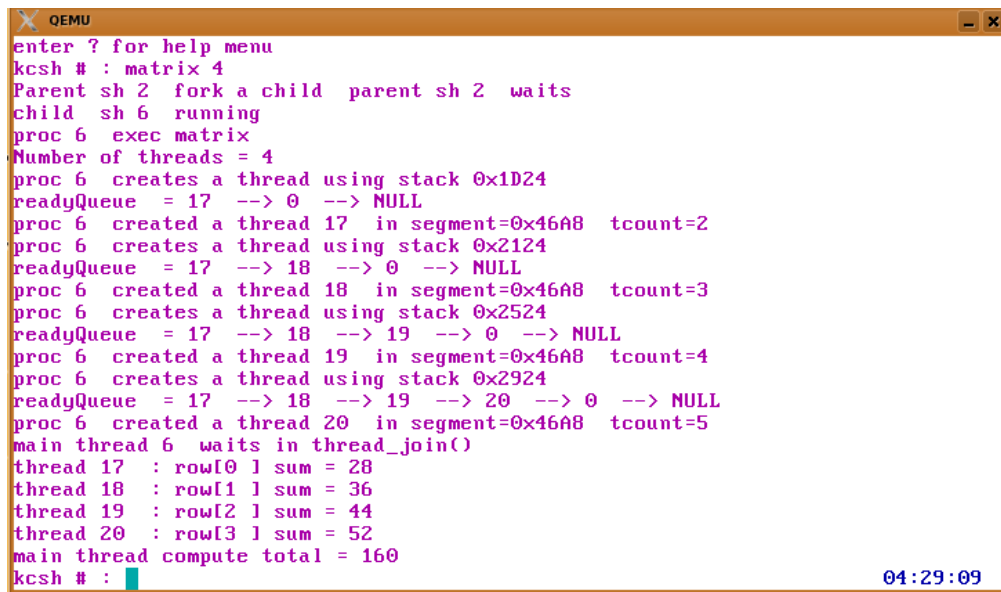
main thread : MUTEX *mutex = mutex_create();
each thread : compute row sum;
               mutex_lock(mutex);
               read current total;
               current total += row sum;
=====> utswitch()
               write total back
               mutex_unlock(mutex);

```

Then there will be no race condition. This is because `mutex_lock()` allows only one thread to continue. Once a thread begins to update total, any other thread trying to do the same will be blocked by the mutex lock. When the thread that holds the mutex lock finishes, it releases the mutex lock, allowing another thread to do the updating. Since the threads update total one at a time, the final total value will be correct regardless of the execution order of the threads.

## 5.16. MTX5.4: Demonstration of Threads in MTX

MTX5.4 demonstrates threads in MTX. The concurrent programs using threads are `qsort`, `matrix`, `race` and `norace`.



```

QEMU
enter ? for help menu
kcsh # : matrix 4
Parent sh 2 fork a child parent sh 2 waits
child sh 6 running
proc 6 exec matrix
Number of threads = 4
proc 6 creates a thread using stack 0x1D24
readyQueue = 17 --> 0 --> NULL
proc 6 created a thread 17 in segment=0x46A8 tcount=2
proc 6 creates a thread using stack 0x2124
readyQueue = 17 --> 18 --> 0 --> NULL
proc 6 created a thread 18 in segment=0x46A8 tcount=3
proc 6 creates a thread using stack 0x2524
readyQueue = 17 --> 18 --> 19 --> 0 --> NULL
proc 6 created a thread 19 in segment=0x46A8 tcount=4
proc 6 creates a thread using stack 0x2924
readyQueue = 17 --> 18 --> 19 --> 20 --> 0 --> NULL
proc 6 created a thread 20 in segment=0x46A8 tcount=5
main thread 6 waits in thread_join()
thread 17 : row[0 ] sum = 28
thread 18 : row[1 ] sum = 36
thread 19 : row[2 ] sum = 44
thread 20 : row[3 ] sum = 52
main thread compute total = 160
kcsh # :
04:29:09

```

### Figure 5.19. Sample Outputs of MTX5.4

Figure 5.19 shows the sample outputs of the concurrent matrix computation program using 4 threads. As the figure shows, all the threads execute in the same segment address of a process. The reader may also test other threads programs, especially race and norace, and compare their outputs.

### Problems

1. Rewrite `int80h()` and `goUmode()` by using `pusha` and `popa` instructions to save and restore the CPU general registers. Then, show how to initialize the Umode stack when creating a process to run in Umode.
2. Rewrite `int80h()` in the assembly file to do the following:
  - (1). Instead of saving Umode registers in `ustack`, save them in the process `PROC` structure.
  - (2). Show the corresponding code for `goUmode()`.
  - (3). Discuss the advantages and disadvantages of this scheme.
3. Redesign the syscall interface to pass syscall parameters in CPU registers
  - (1). Implement syscall (a,b,c,d) as

```
_syscall:
    ! put syscall parameters a,b,c,d in CPU registers ax,bx,cx,dx, respectively, then issue
    INT 80
    ret
```
  - (2). Re-write the syscall handler in C as

```
int kcinth(int ka, int kb, int kc, int kd){.....}
```

which receives a,b,c,d as parameters.
4. Assume: the syscall `int get_name(char *myname)` gets the running `PROC`'s name string and returns the length of the name string. Implement the `get_name()` syscall and test it under MTX5.0.
5. On some CPUs designed for kernel-user mode operations, the CPU has two separate stack pointers; a `ksp` in kernel mode and a `usp` in user mode. When an interrupt occurs, the CPU enters kernel mode and pushes [SR,PC] of the interrupted point into kernel stack. Assume such a CPU. Show how to initialize the kernel stack of a newly created process for it to begin execution from `VA=0` in user mode.
6. Assume: In MTX, P1 is a Casanova process, which hops, not from bed to bed, but from segment to segment. Initially, P1 runs in the segment 0x2000. By a `hop(u16 segment)` syscall, it enter kernel to change segment. When it returns to Umode, it returns to an IDENTICAL Umode image but in a different segment, e.g. 0x4000. Devise an algorithm for the `hop()` syscall, and implement it in MTX to satisfy the lusts of Casanova processes.
7. When a Casanova process tries to change its Umode segment, the target segment may already be occupied by another process. If so, the original process must be evicted to

make room for the Casanova process. Devise a technique that would allow the Casanova process to move into a segment without destroying the original process in that segment.

8. In a swapping OS, a process Umode image may be swapped out to a disk to make room for another process. When a swapped out process is ready to run, its image is swapped in by a swapping process again. In the MTX kernel, P0 can be designated as the swapping process. Design an algorithm to make MTX a swapping OS.

9. What would happen if a process executing a.out issues another `exec("a.out")`?  
`main() { exec("a.out"); }`

10. Both Unix and MTX support the system calls `fork()`, `exec()` and `wait()`, each returns -1 if the syscall fails. Assume that a.out is the binary executable of the following program.

```
main(int argc, char *argv[ ])
{
    if (fork()==0)
        A:  exec("a.out again");
    else
        B:  wait(&argc);
}
```

- (1). Start from a process, e.g. P2. After `fork()`, which process executes the statement A and what does it do? Which process executes the statement B and what does it do?
- (2). Analyze the run-time behavior of this program. If you think it would cause MTX to crash, think carefully again.
- (3). Run a.out under MTX and compare the results with your analysis

11. (1). Modify `exec()` as `execv(char *filename, char *argv[ ])` as in Unix/Linux.  
(2). Assume: `char *env="PATH=bin HOME=/"` is an "environment string" in Umode. Modify `exec()` so that every Umode `main()` function can be written as  
`main(int argc, char *argv[ ], char *env[ ])`

12. In IBM MVS, `pid = create(filename, arg_list);` creates a child process, which executes `filename` with `arg_list` as parameters in a single operation.

- (1). Design an algorithm for the `create()` operation.
- (2). Compare `create()` with `fork()/exec()` of Unix; discuss their advantages and disadvantages.

13. The MTX kernel developed in this chapter can only run 8 processes in user mode. This is because each process is assigned a unique 64KB segment. Assume that each user mode image needs only 32KB space to run. Modify the MTX kernel to support 16 user mode processes, each runs in a unique 32KB memory area.

14. In MTX, the Umode images of all processes are different. Assume that Umode image files have separate I&D spaces. The code of an image file can be loaded to a code segment, and the data+stack can be in a separate segment. Different Umode processes

may execute the same file by sharing the same Code segment. Each process still has its own data+stack segment. Redesign the MTX kernel to allow processes to share "common code".

15. Pthreads programming: Implement the matrix operations and qsort using Pthreads of Linux.

16. Pthreads support a barrier-wait operation, which allows a set of threads to wait until a specified number of threads have called the wait() function of a barrier. Implement the barrier-wait operation in MTX.

## References

- 1 Buttlar,D, Farrell, J, Nichols, B., "PThreads Programming, A POSIX Standard for Better Multiprocessing", O'Reilly Media, 1996
2. Goldt, S., van der Meer, S. Burkett, S. Welsh, M. The Linux Programmer's Guide, Version 0.4. March 1995.
3. IBM MVS Programming Assembler Services Guide, Oz/OS V1R11.0, IBM
4. POSIX.1C, Threads extensions, IEEE Std 1003.1c, 1995
5. Pthreads: <https://computing.llnl.gov/tutorials/pthreads>
6. The Linux Man page Project: <https://www.kernel.org/doc/man-pages>

