

Chapter 4. A Simple Operating System Kernel

Abstract: Chapter 4 starts to develop an operating system (OS) kernel for process management. Rather than presenting a complete kernel in one step, it develops the kernel in incremental steps. First, it uses a simple program to introduce the process concept and demonstrate the principle of context switching. It uses a process life-cycle model to create processes and control their executions in a multitasking environment. It extends the multitasking system to support dynamic process creation, process scheduling and process termination. It shows how to stop and continue a process and extends stop/continue to sleep/wakeup operations for process synchronization and explains their usage in an OS kernel. Then it implements the wait operation to allow parent process to wait for child process termination. It also shows how to adjust process priorities for priority-based process scheduling. Then it integrates these concepts and principles to implement an OS kernel for process management. In each step, it demonstrates the design principle and implementation techniques by a working sample system, which allows the reader to test and observe the internal operations of an OS kernel.

In this chapter, we start to develop a simple operating system kernel for process management. We begin in 16-bit real-mode [Antonakos, 1999] for several reasons. First, the operating environment of Intel x86 based PCs in protected mode [Intel i486, 1990], is quite complex. If we begin in protected mode, it may take a long time to cover the needed background information. Most readers, especially beginners to operating systems, would be overwhelmed by the low level details and may lose interest quickly. In contrast, the 16-bit real mode environment is much simpler and easier to understand. It allows us to get a quick start without worrying about the complexity of protected mode operations. Second, before developing our own device drivers in Chapter 10, we shall use BIOS for basic I/O, which is available only in real mode. Third and more importantly, most OS design and implementation issues, such as process management, process synchronization, device drivers, file system and user interface, etc. do not depend on whether the machine is in real mode or protected mode. Their only differences are in the areas of virtual address spaces, process image size, memory protection and exception processing. These can be deferred until the reader has gained enough working experience in a simpler environment. Then the transition from real mode to protected mode would be relatively easy and smooth. This approach can be justified by the following facts. Despite its simplicity, the real-mode MTX is also the basis of all other forms of MTX in 32-bit protected mode, from uniprocessor systems to symmetric multiprocessing (SMP) [Intel, 1997]. Rather than presenting a complete kernel in one step, we shall develop the MTX kernel in incremental steps. In each step, we explain the concepts and principles involved and show how to apply them to design and implement the various system components. In addition, we shall demonstrate each step by a sample system, which allows the reader to test and observe the internal operations of an OS kernel. In this chapter, we first explain the principle of multitasking, the concept of processes and illustrate context switching by a simple program. Then we extend the multitasking program to support dynamic process creation, process scheduling and process termination. Based on these, we implement

sleep, wakeup and wait operations for process management. This simple multitasking system serves as the starting point of the MTX kernel.

4.1. Multitasking

In general, multitasking refers to the ability of performing several independent activities at the same time. For example, we often see people talking on their cell phones while driving. In a sense, these people are doing multitasking, although a very bad kind. In computing, multitasking refers to the execution of several independent tasks at the same time. In a uniprocessor system, only one task can execute at a time. Multitasking is achieved by multiplexing the CPU's execution time among different tasks, i.e. by switching the CPU's execution from one task to another. If the switch is fast enough, it gives the illusion that all the tasks are executing simultaneously. This logical parallelism is called concurrency. In a multiprocessor system, tasks can execute on different CPUs in parallel in real time. In addition, each processor may also do multitasking by executing different tasks concurrently. Multitasking is the basis of operating systems. It is also the basis of concurrent programming in general.

4.2. The Process Concept

An operating system is a multitasking system. In an operating system, tasks are also called processes. For all practical purposes, the terms task and process can be used interchangeably. In Chapter 2, we defined an execution image as a memory area containing the execution's code, data and stack. Formally, a process is the execution of an image. It is a sequence of executions regarded by the OS kernel as a single entity for using system resources. System resources include memory space, I/O devices and, most importantly, CPU time. In an OS kernel, each process is represented by a unique data structure, called the Process Control Block (PCB) or Task Control Block (TCB), etc. In MTX, we shall simply call it the PROC structure. Like a personal record, which records all the information of a person, a PROC structure contains all the information of a process. In a single CPU system, only one process can be executing at a time. The OS kernel usually uses a global PROC pointer, running or current, to point at the PROC that is currently executing. In a real OS, the PROC structure may contain many fields and quite large. To begin with, we shall define a very simple PROC structure to represent processes.

```
typedef struct proc{
    struct proc *next;
    int         *ksp;
    int         kstack[1024];
}PROC;
```

In the PROC structure, the next field is a pointer pointing to the next PROC structure. It is used to maintain PROCs in dynamic data structures, such as link lists and queues. The ksp field is the saved stack pointer of a process when it is not executing and kstack is the execution stack of a process. As we expand the MTX kernel, we shall add more fields to the PROC structure later.

4.3.1. A Simple Multitasking Program

```
as86 -o ts.o ts.s      # assemble ts.s into ts.o
bcc -c -ansi t.c       # compile t.c into t.o
as86 -d -o mt.x0 ts.o t.o mt.xlib /usr/lib/bcc/libc.a #link
```

```
mount -o loop FImage /mnt; cp mtz0 /mnt/boot/mtz0; umount /mnt
```

```
qemu -fda FImage -no-fd-bootchk
```

[illegible]

At (1), it lets running point to proc0, as shown on the right-hand side of Figure 4.1. Since we assume that running always points at the PROC of the current executing process, the system is now running the process proc0.

At (2), it calls tswitch(), which saves the return address, rPC, in stack.

At (3), it executes the SAVE part of tswitch(), which saves CPU registers into stack and saves the stack pointer sp into proc0.ksp.

At (4), it calls scheduler(), which sets running to point at proc0 again. For now, this is redundant since running already points at proc0. Then it executes the RESUME part of tswitch().

At (5), it sets sp to proc0.ksp, which is again redundant since they are already the same. Then it pops the stack, which restores the saved CPU registers.

At (6), it executes ret at the end of RESUME, which returns to the calling place of tswitch().

4.3.2. Context Switching

Besides printing a few messages, the program seems useless since it does practically nothing. However, it is the basis of all multitasking programs. To see this, assume that we have another PROC structure, proc1, which called tswitch() and executed the SAVE part of tswitch() before. Then proc1's ksp must point to its stack area, which contains saved CPU registers and a return address from where it called tswitch(), as shown in Figure 4.2.

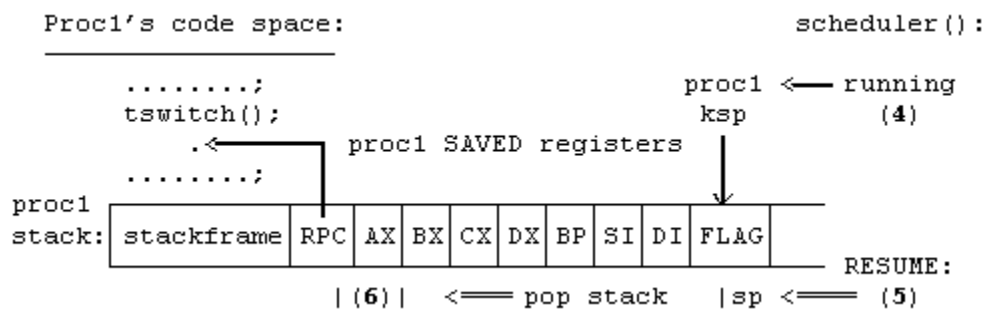


Figure 4.2. Execution Diagram of Proc1

In scheduler(), if we let running point to proc1, as shown in the right-hand side of Figure 4.2, the RESUME part of tswitch() would change sp to proc1's ksp. Then the RESUME code would operate on the stack of proc1. This would restore the saved registers of proc1, causing proc1 to resume execution from where it called tswitch() earlier. This changes the execution environment from proc0 to proc1.

Context Switching:

Changing the execution environment of one process to that of another is called context switching, which is the basic mechanism of multitasking.

4.3.3. A Simple Multitasking System

The scheduler() function simply changes running to running->next. So P0 switches to P1. P1 begins by executing the RESUME part of tswitch(), causing it to resume to the body() function. While in body(), the running process prints its pid and prompts for an input char. Then it calls tswitch() to switch to the next process, etc. Since the PROCs are in a circular link list, they will take turn to run. The assembly code is the same as before, except for the initial stack pointer, which is set to proc[0]'s kstack when execution begins, as in

```
.globl _main, _running, _scheduler
.globl _proc, _procSize ! change proc0 to proc[ ]
mov    sp, #_proc      ! set sp point to proc[0]
add    sp, _procSize    ! let sp point to high end of proc[0]
```

The following lists the C code of the multitasking program.

```
/****** t.c file of a multitasking program *****/
#define NPROC    9          // number of PROCs
#define SSIZE 1024          // proc kstack size = 2KB
typedef struct proc{
    struct proc *next;
    int    *ksp;
    int    pid;              // add pid as proc's ID
    int    kstack[SSIZE];    // proc stack area
}PROC;
PROC proc[NPROC], *running; // define NPROC proc structures
int procSize = sizeof(PROC); // PROC size, needed in assembly code
int body()
{ char c;
  int pid = running->pid;
  printf("proc %d resumes to body()\n", pid);
  while(1){
    printf("proc %d running, enter a key:\n", pid); c=getc();
    tswitch();
  }
}
int init()                  // initialize PROC structures
{
  PROC *p; int i, j;
  For (i=0; i<NPROC; i++){ // initialize all PROCs
    p = &proc[i];
    p->pid = i;              // pid = 0,1,2,..NPROC-1
    p->next = &proc[i+1];    // point to next PROC
    if (i){                  // for PROCs other than P0
      p->kstack[SSIZE-1]=(int)body; // entry address of body()
      for (j=2; j<10; j++) // all saved registers = 0
        p->kstack[SSIZE-j] = 0;
      p->ksp = &(p->kstack[SSIZE-9]); // saved sp in PROC.ksp
    }
  }
  proc[NPROC-1].next = &proc[0]; // all PROCs form a circular list
  running = &proc[0];           // P0 is running;
  printf("init complete\n");
}
int scheduler()              // scheduler() function
{ running = running->next; }
```

```

main()                                // main() function
{
    init();
    while(1){
        printf("P0 running\n");
        tswitch();
    }
}

```

The reader may compile the MTX4.1 program to generate a bootable image. Then boot up and run the program on either a real or virtual PC to observe its run-time behavior. As the system runs, each input key causes a process switch. It uses the process pid to display lines in different colors, just for fun. Figure 4.5 shows the sample outputs of running MTX4.1.

```

init complete
proc 0 running : enter a key :
proc 1 resumes to body()
proc 1 running : enter a key :
proc 2 resumes to body()
proc 2 running : enter a key :
proc 3 resumes to body()
proc 3 running : enter a key :
proc 4 resumes to body()
proc 4 running : enter a key :
proc 5 resumes to body()
proc 5 running : enter a key :
proc 6 resumes to body()
proc 6 running : enter a key :
proc 7 resumes to body()
proc 7 running : enter a key :
proc 8 resumes to body()
proc 8 running : enter a key :
proc 0 running : enter a key :
proc 1 running : enter a key :

```

Figure 4.5. Sample Outputs of MTX4.1

The reader may modify the C code of MTX4.1 to produce different effects. For example, add printf() statements to the scheduler() function to identify the PROCs during context switch. After initialization, instead of calling tswitch(), let P0 call the body() function, etc.

Note that none of the processes, P1 to P8, actually calls body(). What we have done is to convince each process that it called tswitch() to give up CPU just before entering the body() function, and that's where it shall return to when it starts. Thus, we can set up the initial execution environment of a process to control its course of actions. The process has no choice but to obey. This is the power (and joy) of systems programming.

Note also that the processes, P1 to P8, all execute the same code, namely the same body() function. This shows the difference between programs and processes. A program is just a piece of passive code, which has no life in itself. Processes are executions of programs, which make the programs alive. Even when executing the same program each process executes in its own context. For example, in the body() function the local variable

c is in the running proc's stack and pid is the running proc's pid, etc. In a multitasking environment, it is no longer appropriate to describe what a program does. We must look at a program's behavior from the viewpoint of what a process is doing while executing the program. Likewise, an OS kernel is just a (big) program, which is executed by a set of concurrent processes. For convenience, we often say what an OS kernel is doing. But we should really look at an OS kernel from the viewpoint of what the processes are doing while executing the kernel code. The main advantage of this process-based point of view is that it allows us to develop an OS kernel as a set of cooperating processes from ground zero. In fact, this is the major theme of this book, which we shall continue to demonstrate throughout the book.

The assembly function `tswitch()` is the key to understanding context switching and multitasking. A better way to look at `tswitch()` is not as a piece of code but as a relay station where processes change hands, as Figure 4.6 shows.

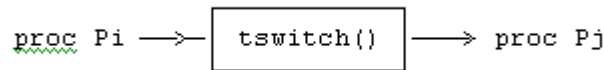


Figure 4.6. Process Switch Box

In this view, `tswitch()` is a switch box where a process `Pi` goes in and, in general, another process `Pj` emerges. In `tswitch()`, the code of SAVE and RESUME are complementary in that whatever items are saved by SAVE, RESUME must restore exactly the same. In principle, a process may save its execution context anywhere, as long as it can be retrieved and restored when the process resumes running again. Some systems save process context in the PROC structure. Since each process already has a stack, we choose to save the process context in the process stack and only save the stack pointer in the PROC structure.

Another way of looking at process creation is as follows. Each process may be thought of as running in a perpetual cycle, in which it runs for a while, then it calls `tswitch()` to give up CPU to another process. Some time later, it regains CPU and emerges from `tswitch()` to resume running again, etc, as shown in Figure 4.7.

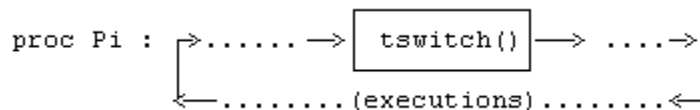


Figure 4.7. Process Execution Cycle

To create a process to run for the first time is the same as to resume a process to run again. We can make a cut in the process execution cycle and inject a suitable condition for it to resume. The natural place to make such a cut is when the process is not running, i.e. immediately after it has executed the SAVE part in `tswitch()` and yielded CPU to another process. At the cut point, we can fabricate a stack frame as if it was saved by the

process itself. When the process becomes running again, it will obey the stack frame and resume to wherever we want it to be.

4.4. Dynamic Process Creation and Process Scheduling

Next, we modify the multitasking program as the initial version of the MTX kernel. As we continue to develop the MTX kernel, we shall demonstrate each step by a complete sample system. For ease of identification, we shall label the sample systems MTXx.y, where x is the chapter in which the MTX kernel is developed, and y is the version number. To begin with, we shall extend MTX4.1 to support dynamic process creation and process scheduling by (static) priority. In order to do these, we need to modify the PROC structure by adding more fields to it. For convenience (mainly for ease of reference in assembly code), we assume that ksp is the second entry and the process kernel stack is always at the high end of the PROC structure. Any new PROC fields should be added between the ksp and kstack entries. The CPU in 16-bit real mode push/pop stack in 2-byte units. Any added field should also be in 2-byte units, so that we can access the stack area as an integer array in C. For the MTX4.2 kernel, the added PROC fields are

```
int status    = proc status = FREE|READY|STOP|DEAD, etc.
int priority  = proc scheduling priority
int ppid     = parent pid
```

In addition, the MTX4.2 kernel also maintains a freeList and a readyQueue, where

- . freeList = a singly-linked list containing all FREE PROCs. Initially, all PROCs are in the freelist. When creating a new process we try to allocate a FREE PROC from freeList. When a process terminates, its PROC structure is eventually returned to the freeList for reuse.

- . readyQueue = a priority queue containing PROCs that are READY to run. In the readyQueue, PROCs of the same priority are ordered first-in-first-out (FIFO).

When the MTX4.2 kernel starts in ts.s, the stack pointer is initialized to point to the high end of proc[0].kstack. Then it calls main(), which calls init() to initialize the data structures as follows.

- . Initialize each PROC by assigning pid=PROC index, status=FREE and priority=0.
- . Enter all PROCs into freeList.
- . Initialize readyQueue = 0.
- . Create P0 as the first running process, i.e. allocate proc[0] and let running point to it.

When init() completes, the system is running P0. P0 calls kfork() to create a child process P1 and enters it into readyQueue. When creating a new process the caller is the parent and the newly created process is the child. But as we shall see shortly, the parent-child process relation is not permanent. The parent of a process may change if its original parent terminates first. Every newly created process has priority=1 and begins execution from the same body() function. After creating P1, P0 enters a while(1) loop, in which it calls tswitch() whenever readyQueue is not empty. Since P1 is already in readyQueue, P0

switches to run P1. At this point, process scheduling is very simple. Among the processes, P0 has the lowest priority 0. All other processes have priority 1. This implies that P0 is always the last PROC in readyQueue. Since PROCs of the same priority are ordered FIFO, other PROCs will take turn to run. P0 will run again if and only if all other PROCs are not runnable. Process scheduling with adjustable priorities will be discussed later in Section 4.4. In the body() function, let the running process print its pid and prompt for a input char, where 's' is to switch process and 'f' is to create a child process. In the MTX kernel, the queue and list manipulation functions are:

```
PROC *get_proc(PROC **list) : return a FREE PROC pointer from list
int  put_proc(PROC **list, PROC *p) : enter p into list
int  enqueue(PROC **queue, PROC *p) : enter p into queue by priority
PROC *dequeue(PROC **queue) : return first element removed from queue
printList(char *name, PROC *list) : print name=list contents
```

Since these functions will be used in all MTX kernels, we precompile them into .o files and put them in a link library, mtplib, for linking. As a programming exercise, the reader is strongly encouraged to implement these functions in a queue.c file and include it in the C code. During linking, the linker will use the reader defined functions instead of those in the mtplib library. This is left as an exercise. The following lists the C code of the MTX4.2 kernel.

```
/****** MTX4.2 kernel t.c file *****/
#define NPROC 9
#define SSIZE 1024
/***** PROC status *****/
#define FREE 0
#define READY 1
#define STOP 2
#define DEAD 3
typedef struct proc{
    struct proc *next;
    int *ksp;
    int pid; // add pid for identify the proc
    int ppid; // parent pid;
    int status; // status = FREE|READY|STOPPED|DEAD, etc
    int priority; // scheduling priority
    int kstack[SSIZE]; // proc stack area
}PROC;
PROC proc[NPROC], *running, *freeList, *readyQueue;
int procSize = sizeof(PROC);
// #include "io.c" // include I/O functions based on getc()/putc()
// #include "queue.c" // implement your own queue functions
int body()
{
    char c;
    printf("proc %d starts from body()\n", running->pid);
    while(1){
        printList("freelist ", freeList); // optional: show the freeList
        printList("readyQueue", readyQueue); // show the readyQueue
        printf("proc %d running: parent=%d\n", running->pid, running->ppid);
        printf("enter a char [s|f] : ");
        c = getc(); printf("%c\n", c);
```

```

        switch(c){
            case 'f' : do_kfork();    break;
            case 's' : do_tswitch();  break;
        }
    }
}
PROC *kfork() // create a child process, begin from body()
{
    int i;
    PROC *p = get_proc(&freeList);
    if (!p){
        printf("no more PROC, kfork() failed\n");
        return 0;
    }
    p->status = READY;
    p->priority = 1;          // priority = 1 for all proc except P0
    p->ppid = running->pid;   // parent = running
    /* initialize new proc's kstack[ ] */
    for (i=1; i<10; i++)     // saved CPU registers
        p->kstack[SSIZE-i] = 0; // all 0's
    p->kstack[SSIZE-1] = (int)body; // resume point=address of body()
    p->ksp = &p->kstack[SSIZE-9]; // proc saved sp
    enqueue(&readyQueue, p); // enter p into readyQueue by priority
    return p;                // return child PROC pointer
}
int init()
{
    PROC *p; int i;
    printf("init ....\n");
    for (i=0; i<NPROC; i++){ // initialize all procs
        p = &proc[i];
        p->pid = i;
        p->status = FREE;
        p->priority = 0;
        p->next = &proc[i+1];
    }
    proc[NPROC-1].next = 0;
    freeList = &proc[0];    // all procs are in freeList
    readyQueue = 0;
    /***** create P0 as running *****/
    p = get_proc(&freeList); // allocate a PROC from freeList
    p->ppid = 0;              // P0's parent is itself
    p->status = READY;
    running = p;             // P0 is now running
}
int scheduler()
{
    if (running->status == READY) // if running is still READY
        enqueue(&readyQueue, running); // enter it into readyQueue
    running = dequeue(&readyQueue); // new running
}
main()
{
    printf("MTX starts in main()\n");
    init();                // initialize and create P0 as running
    kfork();                // P0 creates child P1
    while(1){               // P0 switches if readyQueue not empty

```

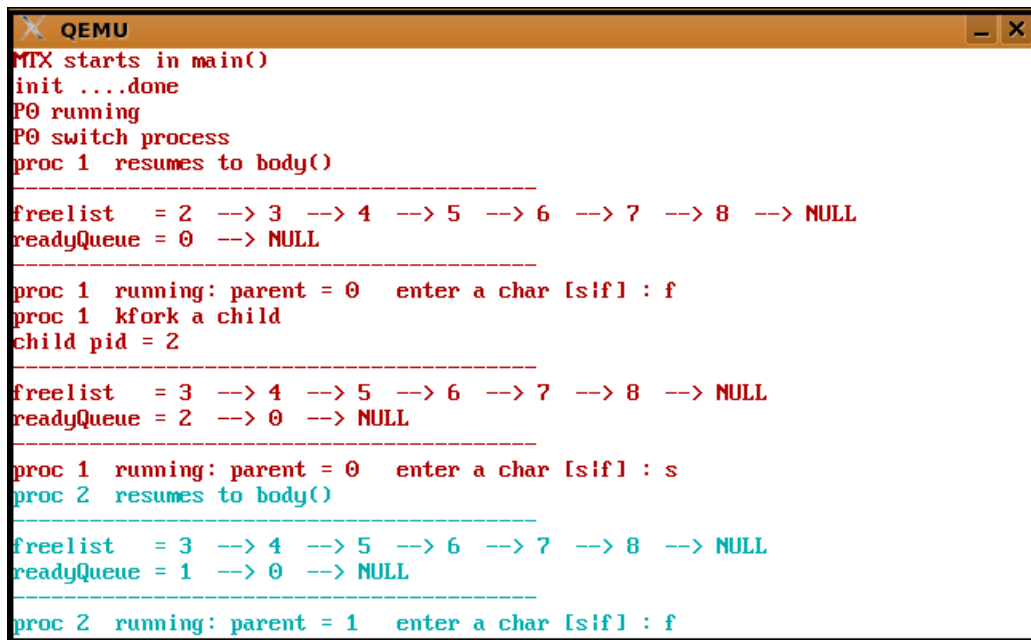
```

        if (readyQueue)
            tswitch();
    }
}
/***** kernel command functions *****/
int do_kfork( )
{
    PROC *p = kfork();
    if (p == 0){ printf("kfork failed\n"); return -1; }
    printf("PROC %d kfork a child %d\n", running->pid, p->pid);
    return p->pid;
}
int do_tswitch(){ tswitch(); }

```

4.5. MTX4.2: Demonstration of Process Creation and Process Switch

MTX4.2 demonstrates dynamic process creation and process scheduling by static priority. Figure 4.8 shows the sample outputs of running the MTX4.2 kernel.



```

QEMU
MTX starts in main()
init ....done
P0 running
P0 switch process
proc 1 resumes to body()
-----
freelist  = 2 --> 3 --> 4 --> 5 --> 6 --> 7 --> 8 --> NULL
readyQueue = 0 --> NULL
-----
proc 1 running: parent = 0  enter a char [sif] : f
proc 1 kfork a child
child pid = 2
-----
freelist  = 3 --> 4 --> 5 --> 6 --> 7 --> 8 --> NULL
readyQueue = 2 --> 0 --> NULL
-----
proc 1 running: parent = 0  enter a char [sif] : s
proc 2 resumes to body()
-----
freelist  = 3 --> 4 --> 5 --> 6 --> 7 --> 8 --> NULL
readyQueue = 1 --> 0 --> NULL
-----
proc 2 running: parent = 1  enter a char [sif] : f

```

Figure 4.8. Sample Outputs of MTX4.2

4.6. Stop and Continue a Process

In the body() function, each command invokes a do_command() function, which calls a corresponding kernel function. The reader may test the kernel functions and observe their executions in action. As an exercise, the reader may modify the MTX4.2 kernel to do the following. In the body() function, add a 'q' command, which lets the running process call do_exit() to become DEAD. The algorithm of do_exit() is

```

do_exit()
{

```

```

    change running PROC's status to DEAD;
    call tswitch() to give up CPU;
}

```

Similarly, add a 't' command, which stops the current running process, and a 'c' command, which lets a stopped process continue. The algorithms of stop and continue are

```

do_stop()
{
    change running PROC.status to STOP;
    call tswitch() to give up CPU;
}
do_continue()
{
    ask for a pid to be continued; validate pid, e.g. 0 < pid < NPROC;
    find the PROC by pid; if PROC.status is STOP, change its status to
    READY and enter it into readyQueue;
}

```

4.7. MTX4.3: Demonstration of stop/continue Operations

Figure 4.9 shows the samples outputs of MTX4.3, which demonstrates stop/continue operations. In the figure, the running process displays its pid, asks for a command char and execute the command, where

's' = switch process, 'f' = kfork a child process, 'q' = to become DEAD
 't' = to become STOP, 'c' = to continue a stopped process by pid

```

QEMU
MTX starts in main()
init ....done
P0 running
P0 switch process
proc 1 resumes to body()
-----
freelist = 2 --> 3 --> 4 --> 5 --> 6 --> 7 --> 8 --> NULL
readyQueue = 0 [0] --> NULL
-----
proc 1 running: priority=1 parent=0 enter a char [sifqitlc] : f
proc 1 kfork a child
child pid = 2
-----
freelist = 3 --> 4 --> 5 --> 6 --> 7 --> 8 --> NULL
readyQueue = 2 [1] --> 0 [0] --> NULL
-----
proc 1 running: priority=1 parent=0 enter a char [sifqitlc] : t
proc 1 stop running
proc 2 resumes to body()
-----
freelist = 3 --> 4 --> 5 --> 6 --> 7 --> 8 --> NULL
readyQueue = 0 [0] --> NULL
-----
proc 2 running: priority=1 parent=1 enter a char [sifqitlc] : c
enter pid to continue : 1
-----
freelist = 3 --> 4 --> 5 --> 6 --> 7 --> 8 --> NULL
readyQueue = 1 [1] --> 0 [0] --> NULL
-----
proc 2 running: priority=1 parent=1 enter a char [sifqitlc] :

```

Figure 4.9. Sample Outputs of MTX4.3

When a process becomes STOP or DEAD, it gives up CPU by calling `tswitch()`. Since the process status is not READY, it will not be entered into `readyQueue`, which makes the process non-runnable. A stopped process becomes runnable again when another process lets it continue. As an exercise, the reader may modify `stop` to a `stop [pid]` operation, which stops a process identified by `pid`. If `pid` is not specified, it stops the running process itself. Similarly, the reader may modify the `continue` operation to let a DEAD process become runnable again. This should convince the reader that if we know the principle of process operations in an OS kernel, we may do anything to them, even perform miracles like resurrection of the dead.

4.8. Sleep and Wakeup Operations

The `stop` operation simply stops a process from running, and the `continue` operation makes a stopped process ready to run again. We can extend `stop` to a `sleep(event)` operation, which stops a running process to wait for an event, and extend `continue` to a `wakeup(event)` operation, which wakes up sleeping processes when their awaited event occurs. `sleep()` and `wakeup()` are the basic mechanism of process synchronization in the Unix kernel. Assume that every PROC has an (added) `event` field. The algorithms of `sleep()` and `wakeup()` are

```
sleep(int event)
{
    record event value in running PROC.event;
    change running PROC.status to SLEEP;
    switch process;
}
wakeup(int event)
{
    for every proc in the PROC array do{
        if (proc.status == SLEEP && proc.event == event){
            change proc.status to READY; // make it READY to run
            enter proc into readyQueue;
        }
    }
}
```

4.9. Sleep and Wakeup Usage

In an OS kernel, `sleep` and `wakeup` are used as follows.

(1). When a process must wait for something, e.g. a resource, identified by an event value, it calls `sleep(event)` to go to sleep, waiting for the event to occur. In `sleep()`, it records the event value in the PROC structure, changes its status to SLEEP and gives up CPU. A sleeping process is not runnable since it is not in the `readyQueue`. It will become runnable again when the awaited event occurs, at which time another process (or an interrupt handler) calls `wakeup(event)` to wake it up.

(2). An event is any value a process may sleep on, as long as another process will issue a wakeup call on the event value. It is up to the system designer to associate each resource with a unique event value. In Unix, event values are usually (global) variable addresses in the Unix kernel, which are unique, so that processes may sleep on distinct event values. wakeup(event) only wakes up those processes that are sleeping on the specified event value. For example, when a process waits for child process termination, it usually sleeps on its own PROC address, which is unique and also known to its children. When a process terminates, it issues wakeup(&parentPROC) to wake up its parent.

(3). Many processes may sleep on the same event, which is natural because many processes may need the same resource that is currently busy or unavailable.

(4). When an event occurs, someone (a process or an interrupt handler) will call wakeup(event), which wakes up ALL the processes sleeping on that event. If no process is sleeping on the event, wakeup() has no effect, i.e. it does nothing. When an awakened process runs, it must try to get the resource again since the resource may already be obtained by another process.

(5). Since an event is just a value, it does not have a memory location to record the occurrence of the event. In order not to miss a wakeup call, a process must go to sleep BEFORE the awaited event occurs. In a uniprocessor system, this is always achievable. In a multiprocessor system, the sleep_first_wakeup_later order cannot be guaranteed because processes may run in parallel in real-time. So sleep/wakeup works only for uniprocessor systems. For multiprocessor systems, we need other kinds of process synchronization tools, e.g. semaphores, which will be discussed later in Chapter 6.

(6). The Unix kernel assigns a fixed priority to a process when it goes to sleep. The assigned priority is based on the importance of the resource the process is waiting for. It is the scheduling priority of the process when it wakes up. If an awakened process has a higher priority than the current running process, process switch does not take place immediately. It is deferred until the current running process is about to exit kernel mode to return to user mode. The reasons for this will also be discussed later.

(7). The assigned priority classifies a sleeping process as either a SOUND sleeper or a LIGHT sleeper. A sound sleeper can only be woken up by its awaited event. A light sleeper can be woken up by other means, which may not be the event it is waiting for. For example, when a process waits for disk I/O, it sleeps with a high priority and should not be woken up by signals. If it waits for an input key from a terminal, which may not come for a long time, it sleeps with a low priority and can be woken up by signals. In the Linux kernel, a sleep process (task) is either INTERRUPTABLE or UNINTERRUPTABLE, which is the same as light or sound sleeper in Unix.

4.10. Implementation of Sleep and Wakeup

For the MTX kernel, we may implement sleep() and wakeup() as follows.


```

int sleep(int event)
{
    running->event = event; // record event in PROC.event
    running->status = SLEEP; // change status to SLEEP
    tswitch();              // give up CPU
}
int wakeup(int event)
{
    int i; PROC *p;
    for (i=1; i<NPROC; i++){ // not applicable to P0
        p = &proc[i];
        if (p->status == SLEEP && p->event == event){
            p->event = 0; // cancel PROC's event
            p->status = READY; // make it ready to run again
            enqueue(&readyQueue, p);
        }
    }
}

```

Strictly speaking, the above implementation has some serious flaws but it works for the current MTX kernel because

- . As of now, MTX is a uniprocessor system, in which only one process can run at any time instant. A process runs until it is ready to switch, e.g. when it sees a 's' or 't' command. Process switch occurs only after a process has completed an operation, never in the middle of an operation. In other words, each process runs alone without any interference from other processes.

- . So far, the MTX kernel does not have any interrupts. When a process runs, it cannot be interfered from any interrupt handler.

Details of how to implement sleep/wakeup and other process synchronization mechanism properly will be covered later in Chapter 6.

4.11. MTX4.4: Demonstration of sleep/wakeup Operations

To demonstrate sleep and wakeup operations, we add the commands 'z' and 'a' to the body() function, where 'z' is for a process to go to sleep on an event value and 'a' wakes up processes, if any, that are sleeping on an event value. Figure 4.10 show the sample outputs of running MTX4.4. As an exercise, the reader may modify MTX4.4 to do the following:

- . Implement a FIFO sleepList containing all SLEEP processes. Modify the sleep() function to enter the sleeping process into the sleepList. Modify the wakeup() function to wake up sleeping processes in order.

- . In the body() function, display sleeping processes and the events they are sleeping on.

. Instead of waking up all processes sleeping an event, wake up only one such process, and discuss its implications.

```

proc 1 running: priority=1 parent=0 enter a char [sif!qiz!a] : f
proc 1 kfork a child
child pid = 2

-----
freelist = 3 --> 4 --> 5 --> 6 --> 7 --> 8 --> NULL
readyQueue = 2 [1] --> 0 [0] --> NULL

-----
proc 1 running: priority=1 parent=0 enter a char [sif!qiz!a] : z
input an event value to sleep: 123
proc 1 going to sleep on event=123
proc 2 resumes to body()

-----
freelist = 3 --> 4 --> 5 --> 6 --> 7 --> 8 --> NULL
readyQueue = 0 [0] --> NULL

-----
proc 2 running: priority=1 parent=1 enter a char [sif!qiz!a] : a
input an event value to wakeup: 123
proc 2 wakeup procs sleeping on event=123
wakeup proc 1

-----
freelist = 3 --> 4 --> 5 --> 6 --> 7 --> 8 --> NULL
readyQueue = 1 [1] --> 0 [0] --> NULL

-----
proc 2 running: priority=1 parent=1 enter a char [sif!qiz!a] :

```

Figure 4.10. Sample Outputs of MTX4.4.

4.12. Process Termination

In an operating system, a process may terminate or die, which is a common term of process termination. As mentioned in Chapter 2, a process may terminate in two possible ways:

- . Normal termination: The process calls `exit(value)`, which issues `_exit(value)` system call to execute `kexit(value)` in the OS kernel, which is the case we are discussing here.
- . Abnormal termination: The process terminates because of a signal. Signals and signal handling will be covered in Chapter 9.

In either case, when a process terminates, it eventually calls `kexit()` in the OS kernel. The general algorithm of `kexit()` is as follows.

4.12.1. Algorithm of `kexit`

```

kexit(int exitValue)
{
    1. erase process user-mode context, e.g. close file descriptors,
       release resources, deallocate user-mode image memory, etc.
    2. dispose of children processes, if any.
    3. record exitValue in PROC.exitCode for parent to get.
    4. become a ZOMBIE (but do not free the PROC)
    5. wakeup parent and, if needed, also the INIT process P1.
}

```

So far, all the processes in MTX run in kernel mode, so they do not have any user mode context yet. User mode will be introduced in Chapter 5. Until then we shall ignore user mode context. So we begin by discussing Step 2 of `kexit()`. In some OS, the execution environment of a process may depend on that of its parent. For example, the child's memory area is within that of the parent, so that the parent process cannot die unless all of its children have died. In Unix, processes only have the very loose parent-child relation but their execution environments are all independent. Thus, in Unix a process may die any time. If a process with children dies first, all the children processes would become orphans. Then the question is: what to do with such orphans? In human society, they would be sent to grandma's house. But what if grandma already died? Following this reasoning, it immediately becomes clear that there must be a process which should not die if there are other processes still existing. Otherwise, the parent-child process relation would soon break down. In all Unix-like systems, the process P1, which is also known as the INIT process, is chosen to play this role. When a process dies, it sends all the orphaned children, dead or alive, to P1, i.e. become P1's children. Following suit, we shall also designate P1 in MTX as such a process. Thus, P1 should not die if there are other processes still existing. The remaining problem is how to implement Step 2 efficiently. In order for a dying process to dispose of orphan children, the process must be able to determine whether it has any child and, if it has children, find all the children quickly. If the number of processes is small, e.g. only a few as in MTX4.4, both questions can be answered effectively by searching all the PROC structures. For example, to determine whether a process has any child, simply search the PROCs for any one that is not FREE and its `ppid` matches the process `pid`. If the number of processes is large, e.g. in the order of hundreds or even thousands, this simple search scheme would be too slow. For this reason, most large OS kernels keep track of process relations by maintaining a process family tree.

4.12.2. Process Family Tree

Typically, the process family tree is implemented as a binary tree by a pair of child and sibling pointers in each PROC, as in

```
struct proc *child, *sibling, *parent;
```

where `child` points to the first child of a process and `sibling` points to a list of other children of the same parent. For convenience, each PROC also uses a parent pointer pointing to its parent. As an example, the process tree shown on the left-hand side of Figure 4.11 can be implemented as the binary tree shown on the right-hand side of Figure 4.11, in which each vertical link is a child pointer and each horizontal link is a sibling pointer. For the sake of clarity, parent and null pointers are not shown.

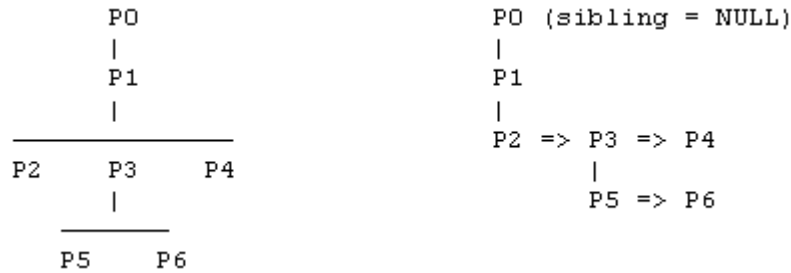


Figure 4.11. Process Tree and Binary Tree

With a process tree, it is much easier to find the children of a process. First, follow the child pointer to the first child PROC. Then follow the sibling pointers to traverse the sibling PROCs. To send all children to P1, simply detach the children list and append it to the children list of P1 (and change their ppid and parent pointer also). Because of the small number of PROCs, MTX does not implement the process tree. This is left as a programming exercise in the Problem section. In either case, it should be fairly easy to implement Step 2 of kexit().

Each PROC has a 2-byte exitCode field, which records the process exit status. In MTX, the low byte of exitCode is the exitValue and the high byte is the signal number that caused the process to terminate. Since a process can only die once, only one of the bytes has meaning. After recording exitValue in PROC.exitCode, the process changes its status to ZOMBIE but does not free the PROC. Then the process calls kwaitup(event) to wake up its parent, where event must be the same unique value used by both the parent and child processes, e.g. the address of the parent PROC structure or the parent pid. It also wakes up P1 if it has sent any orphans to P1. The final act of a dying process is to call tswitch() for the last time. After these, the process is essentially dead but still has a dead body in the form of a ZOMBIE PROC, which will be buried (set FREE) by the parent process through the wait operation.

4.13. Wait for Child Process Termination

At any time, a process may call the kernel function

```
pid = kwait(int *status)
```

to wait for a ZOMBIE child process. If successful, the returned pid is the ZOMBIE child's pid and status contains the exitCode of the ZOMBIE child. In addition, kwait() also releases the ZOMBIE PROC back to the freeList. The algorithm of kwait is

```

int kwait(int *status)
{
    if (caller has no child)
        return -1 for error;
    while(1){ // caller has children
        search for a (any) ZOMBIE child;
        if (found a ZOMBIE child){
            get ZOMBIE child pid
            copy ZOMBIE child exitCode to *status;

```

```

        bury the ZOMBIE child (put its PROC back to freeList)
        return ZOMBIE child pid;
    }
    /*** has children but none dead yet ***
    ksleep(running); // sleep on its PROC address
}
}

```

In the kwait algorithm, the process returns -1 for error if it has no child. Otherwise, it searches for a ZOMBIE child.. If it finds a ZOMBIE child, it collects the ZOMBIE child's pid and exitCode, releases the ZOMBIE PROC to freeList and returns the ZOMBIE child's pid. Otherwise, it goes to sleep on its own PROC address, waiting for a child to terminate. Correspondingly, when a process terminates, it must issue kwakeup(parent) to wake up the parent. Instead of the parent PROC address, the reader may verify that using the parent pid should also work. In the algorithm, when the process wakes up, it will find a dead child when it executes the while loop again. Note that each kwait() call handles only one ZOMBIE child, if any. If a process has many children, it may have to call kwait() multiple times to dispose of all the dead children. Alternatively, a process may terminate first without waiting for any dead child. When a process dies, all of its children become children of P1. As we shall see later, in a real system P1 executes in an infinite loop, in which it repeatedly waits for dead children, including adopted orphans. Therefore, in a Unix-like system, the INIT process P1 wears many hats.

- . It is the ancestor of all processes except P0. In particular, it is the grand daddy of all user processes since all login processes are children of P1.
- . It is the head of an orphanage since all orphans are sent to his house and call him Papa.
- . It is the manager of a morgue since it keeps looking for ZOMBIES to bury their dead bodies.

So, in a Unix-like system if the INIT process P1 dies or gets stuck, the system would stop functioning because no user can login again and the system will soon be full of rotten corpses.

4.14. Change Process Scheduling Priority

In an OS kernel, processes compete for CPU time by scheduling priority. However, process scheduling by priority is meaningful only if process priority can be changed dynamically. Factors that affect process priority typically involve some measure of time. For example, when a process is scheduled to run, it is given a certain amount of time to run. If a process uses up its allotted time whenever it runs, its priority should decrease. If a process gives up CPU before its allotted time expires, the remaining time may be credited to the process, allowing its priority to increase. If a ready process has not run for a long time, its priority should also increase, etc. Since the MTX kernel does not yet have a timer (timer will be introduced in Chapter 8), it is not possible to adjust process priorities dynamically. However, we can simulate dynamic process priority and observe its effect by the following means.

- (1). Add a time field to the PROC structure to simulate process time quantum. When a process is scheduled to run, set its time quantum to a limit value, e.g. 5. While a process runs, decrement its time by 1 for each command it executes. When the simulated time reaches 0, switch process. If a process switches before its time expires, use the remaining time to adjust its scheduling priority. This would simulate process scheduling by time slice and dynamic priority. Implementation of such a scheme is left as an exercise in the Problem section.
- (2). Alternatively, we may implement a change priority operation, which changes the priority of a process. Changing process priority may reorder the readyQueue, which in turn may trigger a process switch.

4.15. MTX4.5: Demonstration of Process Management

In the above sections, we have discussed the concepts and principles of process management in an OS kernel. These include context switching, process creation, process termination, process goes to sleep, wake up processes, wait for child process termination and change process priority. In the sample system MTX4.5, we integrate these concepts and principles and implement the algorithms to demonstrate process management in an OS kernel. The following lists the C code of the MTX4.5 kernel. In order to make the code modular, related functions are implemented in separate files for easier update and maintenance. For example, type.h contains system constants and data structure types, wait.c contains sleep/wakeup, exit and wait functions, kernel.c contains kernel command functions, etc.

```

/***** type.h file *****/
#define NPROC      9
#define SSIZE 1024
#define FREE       0
#define READY      1
#define RUNNING    2    // for clarity only, not needed or used
#define STOPPED    3
#define SLEEP      4
#define ZOMBIE     5
typedef struct proc{
    struct proc *next;
    int *ksp;
    int pid;           // process ID number
    int status;        // status = FREE|READY|RUNNING|SLEEP|ZOMBIE
    int ppid;          // parent pid
    struct proc *parent; // pointer to parent PROC
    int priority;
    int event;         // sleep event
    int exitCode;      // exit code
    int kstack[SSIZE];
}PROC;

/***** wait.c file *****/
int ksleep(int event) { // shown above }
int kwakeup(int event){ // shown above }
int ready(PROC *p) { p->status=READY; enqueue(&readyQueue, p); }
int kexit(int exitValue)
{

```

```

int i, wakeupP1 = 0;
if (running->pid==1 && nproc>2){ // nproc = number of active PROCs
    printf("other procs still exist, P1 can't die yet\n");
    return -1;
}
/* send children (dead or alive) to P1's orphanage */
for (i = 1; i < NPROC; i++){
    p = &proc[i];
    if (p->status != FREE && p->ppid == running->pid){
        p->ppid = 1;
        p->parent = &proc[1];
        wakeupP1++;
    }
}
/* record exitValue and become a ZOMBIE */
running->exitCode = exitValue;
running->status = ZOMBIE;
/* wakeup parent and also P1 if necessary */
kwakeup(running->parent); // parent sleeps on its PROC address
if (wakeupP1)
    kwakeup(&proc[1]);
tswitch(); // give up CPU
}
int kwait(int *status) // wait for ZOMBIE child
{
    PROC *p; int i, hasChild = 0;
    while(1){
        for (i=1; i<NPROC; i++){ // search PROCs for a child
            p = &proc[i]; // exclude P0
            if (p->status != FREE && p->ppid == running->pid){
                hasChild = 1; // has child flag
                if (p->status == ZOMBIE){ // lay the dead child to rest
                    *status = p->exitCode; // collect its exitCode
                    p->status = FREE; // free its PROC
                    put_proc(&freeList, p); // to freeList
                    nproc--; // once less processes
                    return(p->pid); // return its pid
                }
            }
        }
        if (!hasChild) return -1; // no child, return ERROR
        ksleep(running); // still has kids alive: sleep on PROC address
    }
}
}
/***** kernel.c file *****/
int do_tswitch() { // same as in MTX4.4 }
int do_kfork() { // same as in MTX4.4 }
int do_exit() { kexit(0); }
int do_stop() { // same as in MTX4.4 }
int do_continue(){ // same as in MTX4.4 }
int do_sleep() { // same as in MTX4.4 }
int do_wakeup() { // same as in MTX4.4 }
// added scheduling functions in MTX4.5
int reschedule()
{
    PROC *p, *tempQ = 0;
    while ( (p=dequeue(&readyQueue)) ){ // reorder readyQueue

```

```

        enqueue(&tempQ, p);
    }
    readyQueue = tempQ;
    rflag = 0;                                // global reschedule flag
    if (running->priority < readyQueue->priority)
        rflag = 1;
}
int chpriority(int pid, int pri)
{
    PROC *p; int i, ok=0, reQ=0;
    if (pid == running->pid){
        running->priority = pri;
        if (pri < readyQueue->priority)
            rflag = 1;
        return 1;
    }
    // if not for running, for both READY and SLEEP procs
    for (i=1; i<NPROC; i++){
        p = &proc[i];
        if (p->pid == pid && p->status != FREE){
            p->priority = pri;
            ok = 1;
            if (p->status == READY) // in readyQueue==> redo readyQueue
                reQ = 1;
        }
    }
    if (!ok){
        printf("chpriority failed\n");
        return -1;
    }
    if (reQ)
        reschedule(p);
}
int do_chpriority()
{
    int pid, pri;
    printf("input pid ");
    pid = geti();
    printf("input new priority ");
    pri = geti();
    if (pri<1) pri = 1;
    chpriority(pid, pri);
}
int body()
{
    char c;
    while(1){
        if (rflag){
            printf("proc %d: reschedule\n", running->pid);
            rflag = 0;
            tswitch();
        }
        printList("freelist ", freeList); // show freelist
        printQ("readyQueue", readyQueue); // show readQueue
        printf("proc%d running: priority=%d parent=%d enter a char
            [s|f|t|c|z|a|p|w|q]: ",
            running->pid, running->priority, running->parent->pid );
    }
}

```



```

        c = getc(); printf("%c\n", c);
        switch(c){
            case 's' : do_tswitch();      break;
            case 'f' : do_kfork();        break;
            case 'q' : do_exit();         break;
            case 't' : do_stop();         break;
            case 'c' : do_continue();     break;
            case 'z' : do_sleep();        break;
            case 'a' : do_wakeup();       break;
            case 'p' : do_chpriority();   break;
            case 'w' : do_wait();         break;
            default: printf("invalid command\n"); break;
        }
    }
}
/***** main.c file of MTX4.5 kernel *****/
#include "type.h"
PROC proc[NPROC], *running, *freeList, *readyQueue, *sleepList;
int procSize = sizeof(PROC);
int nproc, rflag;          // number of procs, re-schedule flag
#include "io.c"             // may include io.c and queue.c here
#include "queue.c"
#include "wait.c"           // ksleep(), kwakeup(), kexit(), wait()
#include "kernel.c"        // other kernel functions
int init()
{
    PROC *p; int i;
    for (i=0; i<NPROC; i++){ // initialize all procs
        p = &proc[i];
        p->pid = i;
        p->status = FREE;
        p->priority = 0;
        p->next = &proc[i+1];
    }
    freeList = &proc[0]; proc[NPROC-1].next = 0; // freeList
    readyQueue = sleepList = 0;
    /**** create P0 as running *****/
    p = get_proc(&freeList); // get PROC from freeList
    p->status = READY;
    running = p;
    nproc = 1;
}
int scheduler()
{
    if (running->status == READY)
        enqueue(&readyQueue, running);
    running = dequeue(&readyQueue);
    rflag = 0;
}
main()
{
    printf("MTX starts in main()\n");
    init();          // initialize and create P0 as running
    kfork();         // P0 kfork() P1 to run body()
    while(1){
        while(!readyQueue); // P0 idle loop while readyQueue empty
        tswitch();         // P0 switch to run P1
    }
}

```

```
}
}
```

Figure 4.12 shows the sample outputs of MTX4.5, in which the commands are:

s = switch process, f = fork a child process, q = terminate with an exit value
t = stop running process, c = continue a stopped process
z = running process sleep on an event value, a = wake up processes by event value
p = change process priority, w = wait for a ZOMBIE child

```
proc 1 [1 ] running: parent=0
enter a char [sifiticziaipwiq] : f
proc 1 kfork a child
child pid = 2

-----

freelist   = 3 [0 ] -> 4 [0 ] -> 5 [0 ] -> 6 [0 ] -> 7 [0 ] -> 8 [0 ] -> NULL
readyQueue = 2 [1 ] -> 0 [0 ] -> NULL
sleepList  = NULL

-----

proc 1 [1 ] running: parent=0
enter a char [sifiticziaipwiq] : w
proc 2 resumes to body()

-----

freelist   = 3 [0 ] -> 4 [0 ] -> 5 [0 ] -> 6 [0 ] -> 7 [0 ] -> 8 [0 ] -> NULL
readyQueue = 0 [0 ] -> NULL
sleepList  = 1 [1 ] -> NULL

-----

proc 2 [1 ] running: parent=1
enter a char [sifiticziaipwiq] :
```

Figure 4.12. Sample Outputs of MTX4.5

4.16. MTX Development Environment

The initial development platform of MTX is BCC under Linux. Figure 4.13 shows the development and running environment of MTX. The right-hand side of the figure shows two X-window terminals. X-window#1 is for developing the MTX kernel. X-window#2 is for running the MTX kernel on a virtual machine. The software tools used include Linux editors, such as vi or emacs, the BCC compiler-linker package and the archive utility ar for creating and maintaining link library. After creating the source files, the user may use either a Makefile or a mk script to compile-link the source files into a MTX kernel image. There are two ways to test the MTX kernel. The left-hand side of the figure shows a real x86 PC with bootable devices, e.g. a FD containing a MTX disk image. To test the MTX kernel on a real PC, simply copy the MTX kernel to the disk image as /boot/mtx and then boot up MTX on the PC. Until we add a file system to MTX, the MTX system operates in read-only mode. So a FD-emulation CDROM may also be used as the boot device.

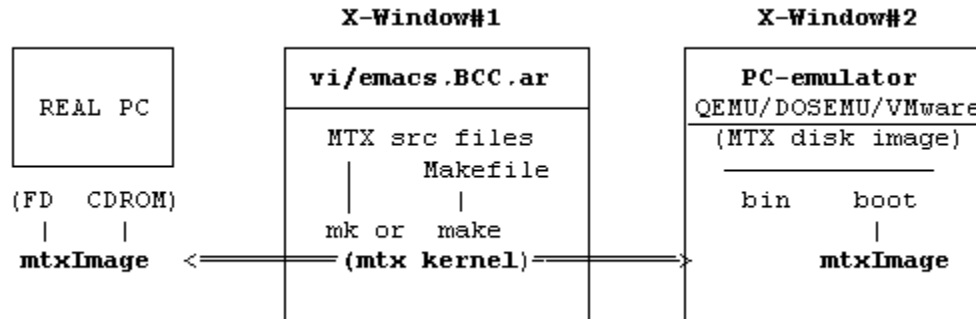


Figure 4.13. Development Environment of MTX.

MTX is intended to be a real operating system. It should work on real PCs, not just in an emulated environment. However, booting from a real PC requires turning off/on a real machine and the booting process is also slow. A faster and more convenient way to run MTX is to use a PC emulator, such as QEMU, DOSEMU or VMware, as shown in the right-hand side box X-Window#2. When running MTX on virtual machines the simplest way is to use the mtximage file as a bootable virtual floppy disk.

```

.QEMU    : qemu -fda mtximage -no-fd-bootchk
.DOSEMU   : configure DOSEMU to use mtximage as virtual FD
.VMware   : configure VM to boot from mtximage as virtual FD
  
```

In all the cases, MTX will boot up and run in a separate window. Thus, we can develop and run MTX without ever leaving the working environment of Linux. Due to its fast turn around time and convenience, most students prefer this method. Alternatively, MTX can also be installed to, and run from, hard disk partitions of virtual machines. The install procedures are described in the Appendix of this book.

Problems

1. The Intel x86 CPU has a pusha instruction, which pushes (in order) all the general registers ax,cx,dx,bx,oldsp,bp,si,di into stack, where oldsp is the sp value before the pusha instruction. Correspondingly, the popa instruction pops the saved registers in reverse order. Assume that tswitch() is modified as

```

_tswitch:
SAVE:    pusha
         pushf
         mov  bx,_running
         mov  2[bx], sp
FIND:    call _scheduler
RESUME:
  
```

- (1). Write assembly code for the RESUME part.
- (2). Corresponding to this tswitch(), show how to initialize the kernel stack of a new process to let it resume to the body() function. Test MTX with the modified tswitch().

2. In addition to general registers, the x86 CPU's segment registers should also be part of the context of a process. As such, they should also be saved and then restored during context switching. Assume that `tswitch()` is modified as

```
_tswitch:
SAVE:  push  ax,bx,cx,dx,bp,si,di  ! in pseudo-code
      pushf
      ! ADD THESE LINES, WHICH SAVE THE CPU'S SEGMENT REGISTERS
      push  ds
      push  ss
      ! END OF ADDED LINES
      mov   bx,_running
      mov   2[bx], sp
FIND:  call _scheduler
RESUME:
```

- (1). Write assembly code for the RESUME part of the modified `tswitch()`.
- (2). Corresponding to the modified `tswitch()`, show how to initialize the kernel stack of a new process. Pay special attention to the contents of `ds` and `ss`, which cannot be arbitrary but must be initialized with some specific values. Test MTX with the modified `tswitch()`.
- (3). The ADDED LINES are not in MTX's `tswitch()` because they are redundant. WHY are they redundant?

2. In the MTX kernel, a process gives up CPU by calling `tswitch()`. Since this is a function call, it is unnecessary to save all the CPU registers. For instance, it is clearly unnecessary to save and restore the AX register. Assume: `tswitch()` is modified as follows.

```
_tswitch:
SAVE:  push  bp                      ! save bp only
      mov   bx,_running
      mov   2[bx], sp
FIND:  call _scheduler
RESUME:
```

- (1). Write assembly code for the RESUME part of the modified `tswitch()`.
- (2). Corresponding to this `tswitch()`, show how to initialize the kernel stack of a new process to let it resume to the `body()` function. Test MTX with the modified `tswitch()`.
- (3). Delete the "push bp" instruction. Repeat (2) and (3). Does MTX still work?
- (4). Why is it necessary to save the bp register of a process? (HINT:stack frames)

3. In MTX, context switching begins when the current running process calls `tswitch()`, which changes the execution to a new process. Some people refer to this as a co-routine call, not a complete context switching. To soothe such critics, we may implement `tswitch()` as

```
_tswitch:  INT 100
          ret
```

When the x86 CPU executes the INT 100 instruction, it first saves the CPU's [FLAG, CS, PC] registers onto the (current) stack. Then, it loads the contents of the memory locations [400, 402] into CPU's [PC, CS], causing it to execute from the newly loaded [PC, CS].

Corresponding to INT 100, the instruction IRET pops 3 items into the CPU's [PC, CS, FLAG] registers. Assume that the memory contents of [400, 402] = [SAVE, 0x1000]. Then, after INT 100, the CPU would execute SAVE in our MTX kernel. Assume: the code of SAVE is:

```
SAVE:  push DS, ES, SS, ax, bx, cx, dx, bp, si, di;
        save sp into running PROC's ksp;
FIND:  call scheduler() to find next running PROC;
-----
RESUME:
```

- (1). Complete the RESUME part, using pseudo-assembly code.
- (2). With this tswitch(), show how to initialize the kstack of a new process for it to begin execution in body().
- (3). Justify whether it is really necessary to implement context switching this way?

4. In MTX, tswitch() calls scheduler(), which essentially picks a runnable process from readyQueue as the next running process. Rewrite tswitch() as tswitch(PROC *current, PROC *next), which switches from the current running process to the next running process.

5. In MTX, there is only one readyQueue, which contains all the processes that are READY to run. A multi-level priority queue (MPQ) consists of, e.g. n+1 priority queues, each at a specific priority level, as shown in the following figure.

```
MPQ  --|-- level_0_queue : lowest priority 0
        |-- level_1_queue : 2nd lowest priority 1
        |-- .....
        |-- level_n_queue : highest priority n
```

In each priority queue, processes have the same priority and are ordered FIFO. Redesign the ready queue in MTX to implement a multi-level priority queue.

6. Implement process family tree for the MTX kernel.

7. Modify the kexit() function to implement the following.

- (1). A dying process must dispose of its ZOMBIE children first.
 - (2). A process can not die until all the children processes have died.
- Discuss the advantages, if any, and disadvantages of these schemes.

8. Linux has a waitpid(int pid, int *status) kernel function, which is equivalent to wait4(int pid, int *status) in Unix. Both allow a process to wait for a specific ZOMBIE child specified by pid. Implement the waitpid()/wait4() function for the MTX kernel.

9. Assume: A bootable MTX kernel image begins with

```
    jmp go          ! a jmp instruction of 2 bytes
    .word kernel_size
go:                  ! assembly code of ts.s
```

where `kernel_size` is the MTX kernel size in bytes, which is the total size of the kernel's code, data and bss sections. When the MTX kernel starts, it can read the `kernel_size` to determine the starting address and size of the initial FREE memory area, which is all the memory area from the end of the MTX kernel to the segment 0x2000. Assume that in the MTX kernel the PROC structure is defined as

```
typedef struct proc{
    struct proc *next;
    int *ksp, pid, status;
    int *kstack; // pointer to per process kernel stack area
}PROC;
PROC *running, *readyQueue;
```

Note that the PROC structure only has a stack pointer but without a stack area, and the MTX kernel does not have an array of PROC structures.

- (1). Implement a `PROC *allocate_proc(sizeof(PROC))` function, which allocates a piece of memory from the FREE memory area as a new proc structure.
- (2). Implement an `int *allocate_stack(SSIZE)` function, which allocates a stack area of SSIZE bytes for an allocated PROC structure.
- (3). When a process dies, it must release the stack area back to the FREE memory. Implement the `free_stack(PROC *p)` function.
- (4). When a process finds a ZOMBIE child, it should release the ZOMBIE PROC back to the FREE memory. Implement the `free_proc(PROC *p)` function.
- (5). Re-write the MTX kernel code using these allocation/deallocation functions for PROCs and process stacks.

10. In the MTX kernel, when a process P_i switches to another process P_j , the kernel stack changes from P_i 's `kstack` to P_j 's `kstack` directly. In many Unix-like systems, the PROC structure of a process and its kernel mode stack are allocated dynamically as in Problem 7. When a process P_i switches to another process P_j , the kernel stack first changes from P_i 's `kstack` to a temporary stack, such as that of an idle process P_0 , and then to P_j 's `kstack`. Why the difference?

11. In the MTX kernel, each process has a unique `pid`, which is the index of its PROC structure. In most Unix-like systems, the `pid` of a process is assigned a unique number in the range [2, 32768]. When the `pid` number exceeds the maximum limit, it wraps around.

- (1). Why are `pid`'s assigned this way?
- (2). Implement this kind of `pid` assignment scheme for the MTX kernel.

12. In the MTX kernel, `kfork()` creates a child process which starts execution from the `body()` function. Assume: the `body()` function is modified as follows.

```
int body(int pid){..... print pid.....}
```

where `pid` is the process `pid`. Furthermore, when a process finishes executing the `body()` function, it returns to `kexit(0)` with a 0 exit value. Modify `kfork()` to accomplish these.

13. In the MTX kernel, `kfork()` creates a child process which starts execution from the `body()` function.

- (1). Assume that a process P_i has executed the following function calls.

body() { int m = 1; A(m); }	int A(int x) { int a = 2; B(a); }	int B(int y) { int b = 3; C(b); }	int C(int z) { int c = 4; HERE: }
--------------------------------------	--	--	--

Draw a diagram to show the stack contents of P_i when execution is at the label HERE in the function C().

(2). Assume that while in the function C(), P_i calls kfork() to create a child process P_j, as shown below.

```
int C(int z){
    int pid = kfork();          // creates a child process Pj
    < -----                // RESUME POINT of Pj
    if (pid==0) {printf("this is the child Pj")
}
```

When the child process P_j runs, it should resume to the same place where it is created, i.e. to the statement pid=kfork() but with a 0 return value. Modify kfork() to accomplish these. Ensure that your kfork() algorithm works for any number of function calls.

HINTS:

- (1). Copy parent's stack to child's stack, add a stack frame for the child to resume.
- (2). Fix up copied stack frame pointers to point to child's stack area.

14. Modify the MTX4.4 kernel to simulate process scheduling by dynamic priority:

- (1). Add a time field to the PROC structure to simulate the CPU use time.
 - (2). When a process (other than P₀) is scheduled to run, give it a time quantum of 5. In the body() function, decrement the running process time by 1 for each command it executes. When the time field reaches 0, switch process. If a process switches before its time expires, add the remaining time to its new priority.
 - (3). In the scheduler() function, if the process is not P₀ and still ready, set its new priority to remaining time + 1, which determines its position in the readyQueue.
- Boot up and run the system to observe process scheduling by dynamic priority.

15. User-level multitasking under Linux:

(a). Given: the following ts.s in 32-bit GCC assembly code:

```
#----- ts.s file -----
.global tswitch, running, scheduler # no underscore prefix
tswitch:
SAVE:    pushal
         pushfl
         movl  running,%ebx
         movl  %esp, 4(%ebx) # integers in GCC are 4 bytes
FIND:    call  scheduler
RESUME:  movl  running, %ebx
         movl  4(%ebx), %esp
         popfl
         popal
         ret
```

(b). The following t.c file

```

/***** t.c file for multitasking under Linux *****/
#include <stdio.h>
#include <stdlib.h> // Linux header files
#include "type.h"   // PROC struct type, same as in MTX4.1
#include "queue.c"  // getproc(), enqueue(), dequeue(); same as in MTX4.1
#define NPROC 9

PROC proc[NPROC], *running, *freeList, *readyQueue;

int body()
{ int c;
  while(1){
    printf("\n*****\n");
    printf("I am task %d My parent=%d\n", running->pid, running->ppid);
    printf("input a char [f|s] : ");
    c = getchar();
    switch(c){
      case 'f': kfork();      break;
      case 's': tswitch();    break;
    }
  }
}

PROC *kfork()
{ PROC *p = get_proc(&freeList);
  if (!p)
    return 0;
  /* initialize the new proc and kstack */
  p->status = READY;
  p->ppid = running->pid;
  p->priority = 1;                // priority = 1
  p->kstack[SSIZE-1] = (int)body; // start to run body()
  p->ksp = &(p->kstack[SSIZE - 1]); // SEE REQUIREMENTS BELOW
  enqueue(&readyQueue, p);
  return p;
}

int init()
{ int i; PROC *p;
  // initialize all PROCs in a freeList, readyQueue=0
}

main()
{
  init();
  kfork(); // create P1
  printf("P0 switch to P1\n");
  tswitch();
}

int scheduler()
{
  if (running->status == READY)
    enqueue(&readyQueue, running);
  running = dequeue(&readyQueue);
}

```


- (1). In the `kfork()` function, there is a line
`p->ksp = &(p->kstack[SSIZE - ?]);`
Fix up the `?` entry to make the `kfork()` code work.
- (2). Under Linux, use `gcc` to compile-link `t.c` and `ts.s` into `a.out`, as in
`gcc -m32 t.c ts.s`
Then run `a.out`.; enter 'f' or 's' and observe the outputs.
- (3). Modify `t.c` to implement `exit()`, `sleep()`, `wakeup()` and `wait()` functions as in MTX4.5.
- (4). The multitasking environment of (3) is the basis of so called user-level threads.
Discuss the similarity and difference between a multitasking kernel, e.g. MTX4.5, and user-level threads.

References

1. Antonakos, J.L., "An introduction to the Intel Family of Microprocessors", Prentice Hall, 1999.
2. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3
3. Intel i486 Processor Programmer's Reference Manual, 1990
4. Intel MP: Intel MultiProcessor Specification, v1.4, 1997

