Advanced Topics in OS: Multithreading

Michael Elhadad

January 22, 1998

1. Threads: Introduction

Material from: *Sun Manual on Networking* 1990.

1. Definition: a thread is an abstraction of a control flow.
2. Difference with process: not associated to address space. Several threads live within a single process.
3. Operations available on threads: scheduling, communication.
4. Good model to manage asynchrony (better than event-based and signals) because no need to disable interrupts during servicing of signal.
5. Important for multi-processor: can schedule threads on several cpus.

2. Functionalities

1. Thread creation, destruction, status, scheduling, suspend, resume.
2. Multiplexing the clock (several threads can sleep concurrently).
3. Monitors and condition variables for synchronization.
4. Message-based synchronization (rendez-vous).
5. Exception handling (notify and escape).
6. Map interrupt to rendez-vous.
7. Map traps into exceptions.
8. Stack management (red-zone protected stacks, stack integrity checks)

3. Thread Creation

1. Threads live in a process that provides an address space.
2. Each thread has its own stack.
3. Each thread is represented by a procedure which is converted to a thread by a call to lwp_create().
4. Threads are scheduled by priority: the highest non-blocking thread is always executing. Threads may block (wait for message or monitor lock).

5. Within a priority, threads run in First-Come/First-Served (FCFS).
6. A set of threads living is the same address space is called a **pod**.
7. A thread is sometimes called a **light-weight process (lwp)**.

Example 1: Create a thread.

```
#include <lwp/lwp.h>
#include <lwp/stackdep.h>

#define MAXPRIO 10

int task();

void main(int arg, char** argv) {
  thread_t tid;

  printf("main here\n");                                   /*1*/
  (void)pod_setmaxpri(MAXPRIO);                            /*2*/
  lwp_setpri(SELF, MAXPRIO);                               /*3*/
  lwp_setstkcache(1000, 2);                                /*4*/
  lwp_create(&tid, task, MAXPRIO, 0, lwp_newstk(), 0);     /*5*/
  printf("main finish\n");                                 /*6*/
}

int task() {
  printf("hello world\n");
  /* now fall through and terminate this thread */
}
```

Compile with:

```
cc -o create1 create1.c -llwp
```

1. main starts as a regular process - prints.
2. the first call to a lwp primitive turns main into a thread. pod_setmaxpri() specifies the highest priority to be used by the scheduler. The range 1..10 becomes available. At the beginning, only MINPRIO is available. Now main() is a thread at priority MINPRIO.
3. lwp_setstkcache() initializes a cache of stacks that can be used to create stacks efficiently. Each call to lwp_newstk() will return a stack of *at least* the size specified (1000 bytes) and the stack is red-zone protected (i.e., access outside of the region will cause a SIGSEGV signal to be generated in hardware).
   The second parameter specifies how big the cache should be initially (number of stacks). A stack allocated from the cache will be automatically freed when the thread dies.
4. We set the priority of main to MAXPRIO too.
5. lwp_newstk() creates a new thread. The thread uses a new stack and runs the task() function at priority MAXPRIO. The task-id is stored into tid. Since main() is also at priority MAXPRIO, task() will not run until main() finishes (FCFS) and the output is:

```
main here
main finish
hello world
```

If we run on a multiprocessor machine, or if we omit line 3, the output can also be:

```
main here
hello world
main finish
```

6. When main() terminates, the corresponding thread dies, and another thread continues. When all threads in a process die, the process dies.
7. Do not confuse lwp_create() and fork(): the new thread lives inside the same process and uses the same file descriptors.

4. Stack Management

1. Need to determine size of stacks. With regular processes not so much an issue because use by default very big stacks, because pages are only allocated as needed.
2. Stacks need to be protected. 2 techniques: red-zone protected and software integrity checking in MMU not available (not important on SUN).

5. Coroutines

1. Threads can be used as pure coroutines - where one thread yields control to another explicitly at the same priority.
2. The function lwp_yield(THREADNULL) causes the current thread to be put at the end of the scheduling queue. If lwp_yield(tid) is called, the thread tid is placed first on the scheduling queue.

Example 2: Coroutines.

```
#include <lwp/lwp.h>
#include <lwp/stackdep.h>

thread_t co1;             /* main tid  */
thread_t co2;             /* coroutine */
thread_t co3;             /* other     */

int coroutine();
int other();

void main(int arg, char** argv) {

  lwp_self(&co1);                                               /*1*/
  lwp_setstkcache(1000, 3);                                     /*2*/
  lwp_create(&co2, coroutine, MINPRIO, 0, lwp_newstk(), 0);     /*3*/
  lwp_create(&co3, other,     MINPRIO, 0, lwp_newstk(), 0);     /*4*/
  printf("1\n");                                                /*5*/
  lwp_yield(THREADNULL);                                        /*6*/
  printf("4\n");                                                /*7*/
```

```
  lwp_yield(co3);                                            /*8*/
  printf("6\n");                                             /*9*/
  exit(0);
}

int coroutine() {
  printf("2\n");
  if (lwp_yield(THREADNULL) < 0) {
    lwp_perror("bad yield");
    return;
  }
  printf("7\n");
  return(0);
}

int other()
{
  printf("3\n");
  lwp_yield(THREADNULL);
  printf("5\n");
  return(0);
}
```

---

1. Get the thread id of main into co1.
2. Prepare stacks.
3. Create thread coroutine.
4. Create thread other.
5. Output will be 1..7 printed in order. All threads have the same MINPRIO priority.

6. Custom Schedulers

1. The following primitives allow the programmer to customize the scheduling algorithm used among threads: lwp_suspend(), lwp_yield(), lwp_resume(), lwp_setpri() and lwp_resched(). lwp_suspend() is useful during debugging, to ensure that a thread is stopped before inspecting it.
2. Use a high-priority thread as a scheduler, all the other threads have a lower priority. The scheduler sleeps for a certain quantum of time, when it wakes up, and reschedules.
3. The following example implements a round-robin time-sliced scheduler.

Example 3: Custom Scheduler.

```
#include <lwp/lwp.h>
#include <lwp/stackdep.h>

#define MAXPRIO 10

int task(int arg);
```

```
int scheduler();

void main(int arg, char** argv) {
  int i;
  pod_setmaxpri(MAXPRIO);
  lwp_setstkcache(1000, 5);                                  /*1*/
  lwp_create(0, scheduler, MAXPRIO, 0, lwp_newstk(), 0);     /*2*/
  for (i=0; i < 3; i++)                                      /*3*/
    lwp_create(NULL, task, MINPRIO, 0, lwp_newstk(), 1, i);
  exit(0);
}

int scheduler() {
  struct timeval quantum;
  quantum.tv_sec = 0;
  quantum.tv_usec = 10000;
  for (;;) {
    lwp_sleep(&quantum);
    lwp_resched(MINPRIO);
  }
}

int task(int arg) {
  for (;;)
    printf("Task %d\n", arg);
}
```

7. Synchronization among Threads

1. 2 types of process synchronization: rendez-vous and monitor.
2. Both are supported by lwp.
3. Rendez-vous is used by the RPC mechanism, works across different address spaces, combines both data transmission and synchronization, and allows easy mapping of asynchronous events into higher level abstractions.
4. monitors are easy to understand (similar to sleep/wakeup in Unix), and more efficient.

8. Rendez-vous

1. The basic primitives are msg_recv() and msg_send(). One thread sends a message, the other receives. Whichever does its side first waits for the other (hence the term rendez-vous). When the rendez-vous takes place, the sender remains blocked until the receiver issues a msg_reply(). Immediately after reply, both threads are unblocked.
2. The sender must provide enough buffer space for the message and the reply. Either one can be empty. While the sender is blocked, the receiver can use both messages, after reply, the receiver decides not to use them anymore (end of transaction).
3. Messages are sent to threads. Each thread has a single queue to receive messages.

4. The receiver specifies the identity of the sending thread it wants to rendez-vous with. It can also specify that any sender is ok. If more precise selection is needed, the client should build a server processes which act as intelligent ports. The id of the sender is supplied to the receiver to this end.

Example 4: Rendez-vous Synchronization.

```
#include <lwp/lwp.h>
#include <lwp/stackdep.h>
#include <lwp/lwperror.h>
#define MAXPRIO 10

thread_t c1, c2;

int sender();
int receiver();

void main(int arg, char** argv) {
  pod_setmaxpri(MAXPRIO);
  lwp_setstkcache(1000, 3);                                  /*1*/
  lwp_create(0, sender, MINPRIO, 0, lwp_newstk(), 0);        /*2*/
  lwp_create(0, receiver, MINPRIO+1, 0, lwp_newstk(), 0);    /*3*/
  exit(0);
}

int sender() {
  char out[20];
  char in[30];
  int i;
  struct timeval wait;
  wait.tv_sec = 2;
  wait.tv_usec = 0;
  for (i = 0; i < 19; i++)
    out[i] = (int)'A'+i;
  out[19] = '\0';
  lwp_sleep(&wait);          /* sleep 2 seconds */
  if (msg_send(c2, out, 20, in, 26) == -1) {
    lwp_perror("msg_send");
    return(-1);
  }
  printf("got reply %s\n", in);
  return(0);
}

int receiver() {
  int i;
  struct timeval wait;
  char *arg, *res;
  int asz, rsz;
  thread_t senderid;
  wait.tv_sec = 1;
  wait.tv_usec = 0;
  senderid = THREADNULL;    /* accept message from anyone */
  if (msg_recv(&senderid, &arg, &asz, &res, &rsz, &wait) == -1) {
    if (lwp_geterr() != LE_TIMEOUT) {
      lwp_perror("msg_recv");
```

```
      return(-1);
    }
    printf("Timeout!\n");
    /* wait forever until a message arrives */
    if (msg_recv(&senderid, &arg, &asz, &res, &rsz, INFINITY) == -1) {
      lwp_perror("msg_recv");
      return(-1);
    }
  }
  printf("Got message: %s\n", arg);
  for (i = 0; i < rsz - 1; i++)
    res[i] = (int)'B'+i;
  res[rsz-1] = '\0';
  msg_reply(senderid);
  printf("Done receiving\n");
  return(0);
}
```