Lecture 4: Process Management

Process Revisited

1. What do we know so far about Linux on X-86?
   - X-86 architecture supports both segmentation and paging.
     - 48-bit logical address goes through the segmentation unit to create
     - 32-bit linear address that goes through the paging unit to create
     - 32-bit physical address.
     - logical address has 16-bit segment selector (13-bit to represent segment and 3 bit to represent priority and location) and 32-bit offset.
     - paging is done using two-level page table (10-bit directory, 10-bit page, 12 bit offset).
   - There are 12 segments reserved for kernel use.
   - There can be at most 4090 processes in X-86 Linux (32-bit version).
   - Each process can have its own Local descriptor (not commonly used) and Task-state segment (TSS).
   - Support four level of privileges (0 to 3).
     - Level 0 is for kernel use.
     - Level 3 is for user.
   - A task_struct is used to represent a process in the kernel.
   - At least, there are two process states: running and blocking. Additional states include ready, new, exit, and suspend.
2. Process control
3. Revisit *proc.c*.
4. Process life-cycle: Assume that you start a text editor in Linux, what happen at the process level?

Race

1. Key terms
   - *Critical section*. A section of code within a process that requires access to share resources and that may not be executed while another process is in a corresponding section of the code.
   - *Race condition*. A situation in which multiple processes or threads read and write a shared data item and the final result depends on the relative timing of their execution (see Figure 1).
   - *Mutual exclusion*. The requirement that when one process is in a critical section, no other process may be in a critical section and access any of those shared resources.
2. Conditions for race avoidance
   (a) Only one process is allowed to enter a critical section at a time.
   (b) It must not be possible for a process requiring access to a critical section to be delayed indefinitely: no deadlock or starvation.
   (c) When no process is in a critical section, any process that requests entry to its critical section must be permitted to enter without delay.
   (d) No assumptions are made about relative process speeds or number of processors.
3. Example: strict alternation (see Figure 2)
4. Semaphore
   - A variable type containing integer counter and a blocking queue
   - Two operations: $down$ and up (see Figure 3)
     – Down: if value = 0, the process goes to sleep without completing the $down$. If value > 0, the operation decrements the value by one and continues. The operation is done **atomically**.
     – UP: the operation increments the value by one. If there are processes waiting to complete earlier $down$ operations, a process is randomly picked to complete the $down$ operation. The steps to wake up a process and increment the semaphore is also done **atomically**.
5. Test-Set-Lock (TSL) (e.g. TSL RX, LOCK)
   - TSL copies the content of $LOCK$ to RX and then set $LOCK$ to a non-zero value (see Figure 4).

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

#define NUMBER_OF_PROCS 2
#define MAX_COUNT 1000000

int sharedCounter=0;

void MyCounterPlus()
{
   int i;
   for (i = 0; i < MAX_COUNT; i++)
      sharedCounter++;
}
void MyCounterMinus()
{
   int i;
   for (i = 0; i < MAX_COUNT; i++)
      sharedCounter--;
}
int main(void)
{
   int i;
   pthread_t thread[NUMBER_OF_PROCS];
   for (i = 0; i < NUMBER_OF_PROCS; i++)     {
      if ((i % 2) == 0)        {
         pthread_create(&thread[i], NULL, (void *) MyCounterPlus, NULL);
      }
      else        {
         pthread_create(&thread[i], NULL, (void *) MyCounterMinus, NULL);
      }

   }
   for (i = 0; i < NUMBER_OF_PROCS; i++)
         pthread_join(thread[i], NULL);
   fprintf (stdout, "sharedCounter = %d\n", sharedCounter);
   return 0;
}
```

Figure 1: An Example of Race Condition using Pthread.

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

#define NUMBER_OF_PROCS 2
#define MAX_COUNT 10000
int sharedCounter = 0;
int turn = 0;

void MyCounterPlus()
{
    int i;
    for (i = 0; i < MAX_COUNT; i++)
    {
        while (turn != 0);
        sharedCounter++;
        turn = 1;
    }
}
void MyCounterMinus()
{
    int i;
    for (i = 0; i < MAX_COUNT; i++)
    {
        while (turn != 1);
        sharedCounter--;
        turn = 0;
    }
}
int main(void)
{
    int i;
    pthread_t thread[NUMBER_OF_PROCS];
    for (i = 0; i < NUMBER_OF_PROCS; i++)     {
        if ((i % 2) == 0)         {
            pthread_create(&thread[i], NULL, (void *) MyCounterPlus, NULL);
        }
        else         {
            pthread_create(&thread[i], NULL, (void *) MyCounterMinus, NULL);
        }
    }
    for (i = 0; i < NUMBER_OF_PROCS; i++)
            pthread_join(thread[i], NULL);
    fprintf (stdout, "sharedCounter = %d\n", sharedCounter);
    return 0;
}
```

Figure 2: Providing Mutual Exclusion with Strict Alternation.

```
/* From Andrew Tanenbaum's Modern Operating Systems, 2nd Edition */
#define N 10000                 /* Buffer size */
typedef int semaphore;
semaphore mutex = 1;            /* Access to critical section */
semaphore empty = N;            /* count available slots */
semaphore full = 0;             /* count occupied slots */

void producer(void)
{
        int item;
        while (1)
        {
                item = produce_item();
                down(&empty);
                down(&mutex);
                insert_item(item);
                up(&mutex);
                up(&full);
        }
}

void consumer(void)
{
        int item;
        while (1)
        {
                down(&full);
                down(&mutex);
                item = remove_item();
                up(&mutex);
                up(&empty);
                consume_item(item);
        }
}
```

Figure 3: Solving Producer and Consumer Problem with Semaphores.

```
; From Andrew Tanenbaum's Modern Operating Systems, 2nd Edition

enter_region:
         TSL RX, LOCK
         CMP RX, R0  ; assume R0 always contains ZERO
         JNE enter_region
         RET         ; return to caller once critical region is entered

leave_region:
         SW R0, MUTEX
         RET

================

; the following code can be used to make MUTEX for user level thread package
; From Andrew Tanenbaum's Modern Operating Systems, 2nd Edition

mutex_lock:
         TSL RX, MUTEX
         CMP RX, R0
         JZE ok
         CALL thread_yield
         JMP mutex_lock
ok:
         RET

mutex_unlock:
         SW R0,MUTEX
         RET
```

Figure 4: Examples of using TSL.