628 Lecture Notes Week 5

Table of Contents

Recap

- So far, we have concentrated on the "what" and "why" of Operating Systems.
- "What": An OS manages the computing systems resources (CPU, memory, devices). It exploits and uses both hardware and software.
- "Why": Efficiency. When there are multiple processes, a process that needs to perform I/O will be "Blocked" until the I/O had been done. In the meantime, other processes can run.
- "How": Lots has YET to be covered. Nonetheless, there are some things we know:
    - The OS maintains a fixed size table of the processes it manages. Each entry in the table contains information such as: the Process ID (PID), its state (READY, RUN, BLOCKED) and the machine register values to restart the process.
    - For each process the OS maintains a fixed size table of *file descriptors*. (Entries 0, 1 and 2 are opened by the OS and correspond to *stdin*, *stdout*, and *stderr*.)

Tables maintained by the Operating System

1. The Process Table

The process table contains information about each process including:

- Its *Process Identification (PID)*: a unique positive integer.
- The PID of its parent process (*PPID).*
- Its current *State* (Running, Blocked or Ready).
    - Note: There may be more specific types of the Blocked state such as "waiting for a semaphore" and other states to be discussed later.
- The limits of its various address spaces including:
    - Its *text* area (where the machine language instructions are located).
    - Its *global data* area.
    - Its *heap* area (for dynamically allocated memory).
    - Its *stack* area (for local variables and subroutine linkage).
    - Its *global read-only data* area.
- The *register values* when the process was last running.
- Various bookkeeping and statistics such as "time the process started", "cumulative cpu time in user mode and system mode", etc.

2. Tables maintained for EACH process

Each process also maintains a table for resources of that process.

The Thread Table

- Each Thread has an id (TID).
- A State (like a process it can be "running", "blocked" or "ready".
- The *register values* when the thread was last running. (Sometimes only the Stack pointer is needed.)
- The *stack* area for the process. (The other memory areas are shared with the process that created it an all other threads of the process.)

The File Table

The file table (of finite size per process) contains slots for all open and available file descriptors.

Each file descriptor for an open file contains:

- Hooks to read, write or close a file.
- The size of the file and the current position,

More about Threads

1. Why threads can be tricky

- While it is easy for threads to interact (since they share the same global variables and open files), the interaction can be tricky.
- For example, if 2 ATMs are accessed concurrently, a separate Thread handles each, both Threads have access to the global variable `account_balance` and a deposit of $100 is made to each when the current balance is $1000, following is possible:
    - Each Thread executes (at the machine language level):

```
loadRegister balance
addToRegister deposit
storeRegister balance
```

If each Thread executes this sequence of instructions without interruption, everything will work and the balance will increased by $200 (i.e. 2 deposits of $100 each.)

- - However, if one Thread is interrupted just after the “loadRegister” instruction, the other Thread will correctly increase the balance by $100. Alas, when the interrupted Thread regains control, it will still think the balance is $1000...

2. Kernel Threads vs. User Threads

- Some OSes support Threads natively but Threads can also be implemented in User space.
- OS calls are more “expensive” than function calls in user space because an *interrupt* (such as a software interrupt or “trap” instruction) is required. This involves saving all registers on the stack and (usually) switching stacks (from user stack to supervisor stack). Typically, OS calls are an order of magnitude more costly than simple function invocations.
- User Threads can be managed in user space. The Thread table needs to hold the register values for each process along with its state and priority. Switching from one Thread to another (the *context switch*) can be done without kernel calls. Similarly, the *scheduler* can be done in user space.
- Similarly, things like timers can be done in user space.
- **However**, since the kernel is completely unaware of the threads (it knows only about the

process that contains the threads) if a single Thread performs an OS call (such as I/O) that puts the process into the Blocked state, then all Threads are prevented from running.

- When the kernel knows about the Threads, a Thread that blocks does not prevent other threads from running. The kernel can also use more sophisticated scheduling algorithms (such as MacOS's *Grand Central Dispatch*).
- Support for Threads is also common in programming languages such as Java, Go, D and many others.
- In the case of Java, code runs in the *Java Virtual Machine* (JVM) which controls all I/O and typical OS calls. So all threads are effectively user level (and hence efficient). The Java language also includes direct support for inter-process communication. (We shall examine Java's mechanisms later.)

Inter-process Communication (IPC)

1. A basic problem: critical sections and atomic operations

- Threads (almost always) use shared resources such as global variables.
- At a minimum, some mechanism is required to ensure that only one thread can perform certain operations at a given time.
- Such as section of code is called a *critical section*.
- Note: we have to think of the program executing machine language, not “lines of C code”.
    - For example, the C code `i = i + 1;` may be translated into a single machine language instruction (such as `increment i`) or several instructions (such as `loadRegister i; addRegister 1; storeRegister i`)
    - If the operation (incrementing the value of i by 1) is done in a single instruction, then it is an *atomic* operation. It is impossible that the instruction begin execution and then control is given to another Thread before the operation completes. (OK, to be more complete, once we look at virtual memory we discover that it is possible for a single instruction to be “interrupted” by a page fault; nonetheless, as we shall see, even then the operation of a single instruction is effectively atomic.)

Using the pthread library

See introduction to pthreads for details on using this package.

Creating threads, running concurrently, no synchronization

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

  #define N_REPEAT 200

  void *print_string(void *ptr);

  int main() {
      pthread_t thread1, thread2;
      char * string1 = "abcd";
      char * string2 = "WXYZ";
      int iret1, iret2;

      /* Create independent threads each of which will execute function */

      iret1 = pthread_create(&thread1, NULL, print_string, (void*) string1);
      iret2 = pthread_create(&thread2, NULL, print_string, (void*) string2);

      /* Wait till threads are complete before main continues. Unless we  */
      /* wait we run the risk of executing an exit which will terminate   */
      /* the process and all threads before the threads have completed.   */

      pthread_join(thread1, NULL);
      pthread_join(thread2, NULL);

      printf("Thread 1 returns: %d\n", iret1);
      printf("Thread 2 returns: %d\n", iret2);
      exit(0);
  }

  void * print_string(void *ptr) {
      char *cp;
      for (int i = 0; i < N_REPEAT; i++) {
          cp = (char *) ptr;
          while (*cp) {
              putchar(*cp);
              fflush(stdout);
              cp++;
          }
          usleep(1);
      }
}
```

1. Using "mkdir" for exclusive access to critical section

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/types.h>
#include <sys/stat.h>

#define N_REPEAT 2000

void *print_string(void *ptr);

int main() {
    pthread_t thread1, thread2;
    char * string1 = "abcd";
    char * string2 = "WXYZ";
    int iret1, iret2;

    /* Create independent threads each of which will execute function */

    iret1 = pthread_create(&thread1, NULL, print_string, (void*) string1);
    iret2 = pthread_create(&thread2, NULL, print_string, (void*) string2);

    /* Wait till threads are complete before main continues. Unless we  */
    /* wait we run the risk of executing an exit which will terminate   */
    /* the process and all threads before the threads have completed.   */

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    printf("Thread 1 returns: %d\n", iret1);
    printf("Thread 2 returns: %d\n", iret2);
    exit(0);
}

void *print_string(void *ptr) {
    char *cp;
    for (int i = 0; i < N_REPEAT; i++) {
        cp = (char *) ptr;
        while (mkdir("junk", 0777) != 0) {
            usleep(100);
        }
        while (*cp) {
            putchar(*cp);
            fflush(stdout);
            cp++;
        }
        rmdir("junk");
    }
}
```

2. Using "pthread_mutex" for exclusive access to critical section

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/types.h>
#include <sys/stat.h>

#define N_REPEAT 2000

void *print_string(void *ptr);
pthread_mutex_t mutex_stdout = PTHREAD_MUTEX_INITIALIZER;

int main() {
    pthread_t thread1, thread2;
    char * string1 = "abcd";
    char * string2 = "WXYZ";
    int iret1, iret2;

    /* Create independent threads each of which will execute function */

    iret1 = pthread_create(&thread1, NULL, print_string, (void*) string1);
    iret2 = pthread_create(&thread2, NULL, print_string, (void*) string2);

    /* Wait till threads are complete before main continues. Unless we  */
    /* wait we run the risk of executing an exit which will terminate   */
    /* the process and all threads before the threads have completed.   */

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    printf("Thread 1 returns: %d\n", iret1);
    printf("Thread 2 returns: %d\n", iret2);
    exit(0);
}

void * print_string(void *ptr) {
    char *cp;
    for (int i = 0; i < N_REPEAT; i++) {
        cp = (char *) ptr;

        pthread_mutex_lock(&mutex_stdout);

        while (*cp) {
            putchar(*cp);
            fflush(stdout);
            cp++;
        }
        pthread_mutex_unlock(&mutex_stdout);
    }
}
```

3. Using "pthread_mutex" with many Threads

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/types.h>
#include <sys/stat.h>

#define N_REPEAT 1000
#define N_THREADS 5

void *print_string(void *ptr);
pthread_mutex_t mutex_stdout = PTHREAD_MUTEX_INITIALIZER;

int main() {
    pthread_t thread[N_THREADS];
    char * strings[] = {"abcd", "WXYZ", "1234", "++++", "------"};
    int iret[N_THREADS];

    /* Create independent threads each of which will execute function */

    for (int i = 0; i < N_THREADS; i++) {
        iret[i] = pthread_create(&thread[i], (void *) 0,
                print_string, (void*) strings[i]);
    }

    /* Wait till threads are complete before main continues. Unless we  */
    /* wait we run the risk of executing an exit which will terminate   */
    /* the process and all threads before the threads have completed.   */
    for (int i = 0; i < N_THREADS; i++) {
        pthread_join(thread[i], NULL);
    }

    printf("All Done\n");
    exit(0);
}

void *print_string(void *ptr) {
    char *cp;
    printf("PID: %d, Thread ID: %d\n", getpid(), pthread_self());
    for (int i = 0; i < N_REPEAT; i++) {
        cp = (char *) ptr;

        pthread_mutex_lock(&mutex_stdout);

        while (*cp) {
            putchar(*cp);
            fflush(stdout);
```

```
              cp++;
          }
          pthread_mutex_unlock(&mutex_stdout);
      }
  }
```

Notes of Lab 4

- Example of using a pipe:

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

  char *cmd1[] = {"ls", 0}; //Output goes to pipe input
  char *cmd2[] = {"/usr/bin/tr", "a-z", "A-Z", 0}; //Input comes from pipe
output

  int
  main(int argc, char **argv) {
      int pid, status;
      int fd[2];

      pipe(fd); //Create a pipe; fd[0] is input, fd[1] is output

      switch (pid = fork()) {

          case 0: /* child */

              /* Using close/dup */
              close(0);
              dup(fd[0]);
              /* Preferred: use dup2*/
              //            dup2(fd[0], 0); //pipe input is now stdin for child
              close(fd[1]); //child does not use pipe output
              execvp(cmd2[0], cmd2); //child executes "tr" command
              perror(cmd2[0]); //SHOULD NOT GET HERE!
              exit(1);

          case -1://SHOULD NOT GET HERE (indicates fork failure)
              perror("fork");
              exit(1);

          default: /* parent */
              dup2(fd[1], 1);
              close(fd[0]); /* the parent does not need this end of the pipe */
              execvp(cmd1[0], cmd1);
              perror(cmd1[0]); //SHOULD NOT GET HERE (exec failure)
              break;


      }
```

```
        exit(0);
    }
```