CS 240
Computer Systems

#8: [Limited] Direct Execution and Threads II

Sept. 21, 2021 · Wade Fagen-Ulmschneider

Program Execution: Direct Execution Model

Operating System		Process + Thread
1. Create entry for process and thread 2. Allocate memory for process and thread 3. Load program into memory 4. Set up stack (argv/argc) 5. Clear registers 6. Call main()	⇒	*(OS has CPU control.)*
(OS does not have CPU control.)		7. Run main() 8. return from main()
9. Free memory for process 10. Remove process from process list	⇐	*(OS has CPU control.)*

What is the problem with this model?

Protection Levels:

We can develop multiple levels of protection so that the process must request access to certain resources from the Operating System. This way, there is a "handoff" between the OS and the application:

Operating System		Process + Thread
1. Process Init 2. return-from-trap	⇒ Save and Clear Registers Set to "user mode"	*(OS has CPU control.)*
(OS does not have CPU control.)		3. Run main() 4. Makes a system call ...calls **trap-to-OS**
5. Trap Handler ...do syscall work... 6. return-from-trap	⇐ Save+Swap Registers Set to "kernel mode"	*(OS has CPU control.)*
(OS does not have CPU control.)	⇒ Save+Swap Registers Set to "user mode"	...execution continues...

Execution: Kernel and User Modes

There are many different "protection levels" in modern systems:

- Ring 0 ("kernel mode"):
- Ring 1 and 2:
- Ring 3 ("user mode"):
- Ring -1 (VT-x):

Trapping to the OS: More than Just System Calls

There are several mechanisms to regain CPU control from an application back to the OS:

1. **System Calls**:
2.
3.

Additional Reading: "Operating Systems: Three Easy Pieces"
Ch. 6: 6 Direct Execution (https://pages.cs.wisc.edu/~remzi/OSTEP/)

Five-State Thread Model

When the operating system has control over the CPU and needs to decide what program to run, it must maintain a model of all threads within the CPU.

We commonly refer to the "state" of a thread as part of the five-state model:

08/thread-count.c

```c
int ct = 0;

void *thread_start(void *ptr) {
  int countTo = *((int *)ptr);

  int i;
  for (i = 0; i < countTo; i++) {
    ct = ct + 1;
  }

  return NULL;
}

int main(int argc, char *argv[]) {
  // Parse Command Line:
  if (argc != 3)  {
    printf("Usage: %s <countTo> <thread count>\n", argv[0]);
    return 1;
  }

  const int countTo = atoi(argv[1]);
  /* [...error checking...] */
  const int thread_ct = atoi(argv[2]);
  /* [...error checking...] */

  // Create threads:
  int i;
  pthread_t tid[thread_ct];
  for (i = 0; i < thread_ct; i++) {
    pthread_create(&tid[i], NULL,
                            thread_start, (void *)&countTo);
  }

  // Join threads:
  for (i = 0; i < thread_ct; i++) {
    pthread_join(tid[i], NULL);
  }

  // Display result:
  printf("Final Result: %d\n", ct);
  return 0;
}
```

Multiple Threads and Synchronization

In the program to the left, we launch a number of different threads that will count up together in parallel!

Q1: What do we expect when we run this program?

Q2: What is the output of this program when it's running as:

`./count 100 2`

Q3: What is the output of this program when it's running as:

`./count 100 16`

Q4: What is the output of this program when it's running as:

`./count 10000000 2`

Q5: What is the output of this program when it's running as:

`./count 10000000 16`

Q6: What is going on???