[Limited] Direct Execution and Threads II

CS 240 - The University of Illinois

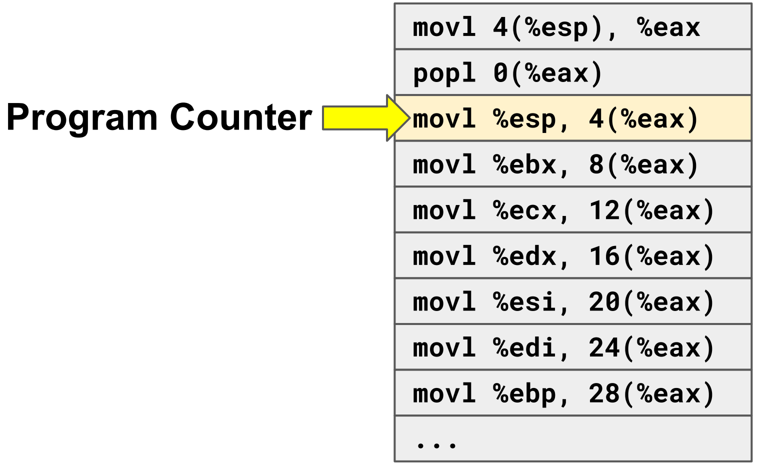
Wade Fagen-Ulmschneider

September 21, 2021

Direct Execution

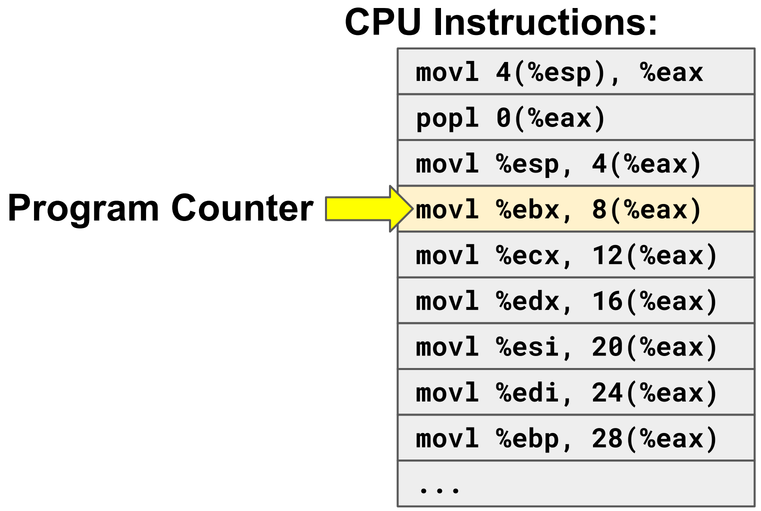
Simple CPU Operation

CPU Instructions:



```
movl 4(%esp), %eax
popl 0(%eax)
movl %esp, 4(%eax)
movl %ebx, 8(%eax)
movl %ecx, 12(%eax)
movl %edx, 16(%eax)
movl %esi, 20(%eax)
movl %edi, 24(%eax)
movl %ebp, 28(%eax)
...
```

Simple CPU Operation



```
movl 4(%esp), %eax
popl 0(%eax)
movl %esp, 4(%eax)
movl %ebx, 8(%eax)
movl %ecx, 12(%eax)
movl %edx, 16(%eax)
movl %esi, 20(%eax)
movl %edi, 24(%eax)
movl %ebp, 28(%eax)
...
```

Direct Execution

OS:

1. Create entry for process
2. Allocate memory for process
3. Load program into memory
4. Set up stack (argv/argc)
5. Clear registers
6. **call** main()

Process:

7. Run main()
8. **return** from main()

OS:

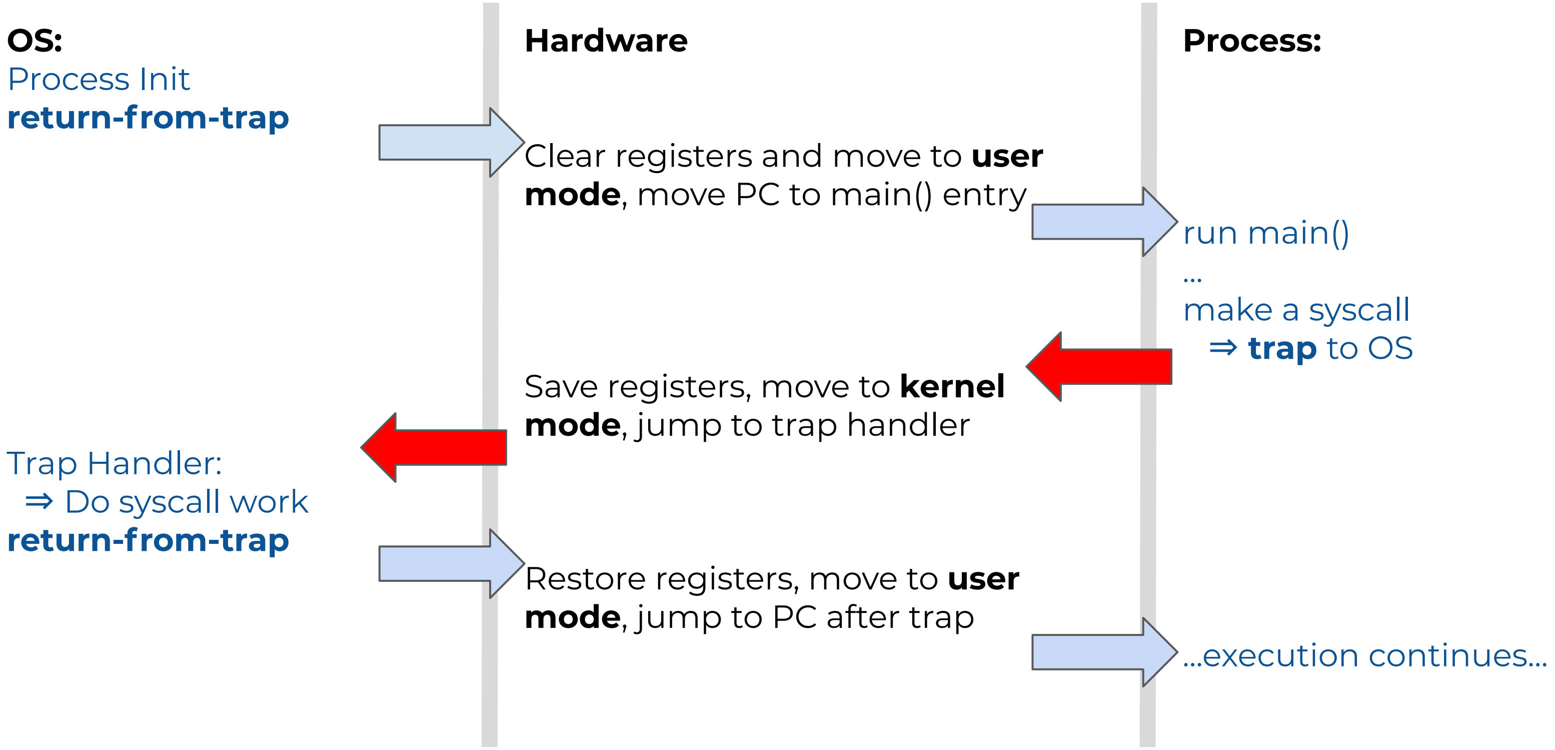
9. Free memory for process
10. Remove process from process list

Direct Execution

Problems?

Protection Levels

Limited Direct Execution



OS:
Process Init
return-from-trap

Hardware
Clear registers and move to **user mode**, move PC to main() entry

Process:
run main()
...
make a syscall
⇒ **trap** to OS

Save registers, move to **kernel mode**, jump to trap handler

Trap Handler:
⇒ Do syscall work
return-from-trap

Restore registers, move to **user mode**, jump to PC after trap

...execution continues...

Facilitate Multiple Applications

Big Idea: Don’t give apps direct access to hardware!

Protection Levels / “Rings”

Ring 0:

Protection Levels / “Rings”

Ring 3:

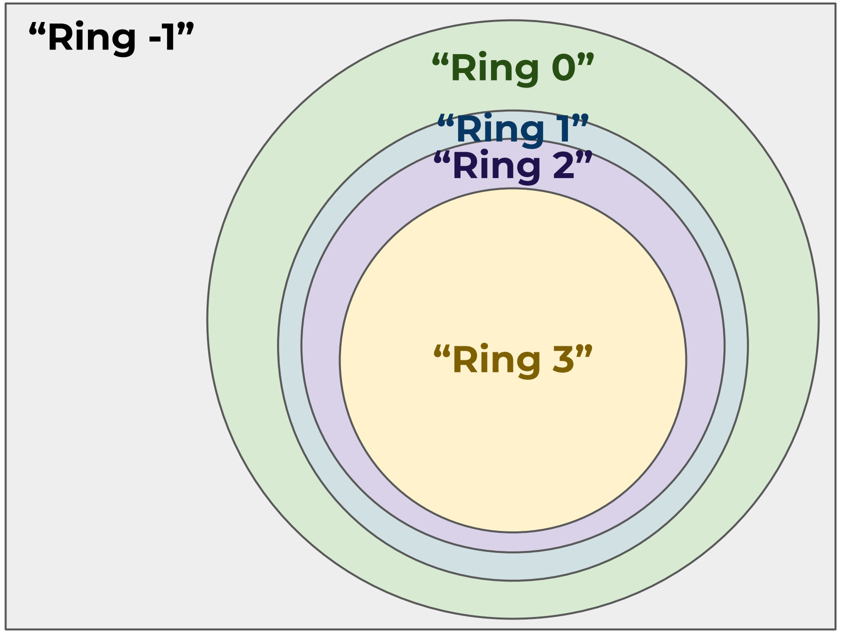
Protection Levels / “Rings”

Rings 1 and 2:

Protection Levels / “Rings”

Ring -1:

Protection Levels / “Rings”



Trapping to the OS

Trap Mechanisms

#1: System Calls

Trap Mechanisms

#2:

Trap Mechanisms

#3:

Five State Process/Thread Model

08/thread-count.c

thread-count.c

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

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;
}

```

thread-count.c

```c
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]);
  if (countTo == 0) { printf("Valid `countTo` is required.\n"); return 1; }

  const int thread_ct = atoi(argv[2]);
  if (thread_ct == 0) { printf("Valid thread count is required.\n"); return 1; }

```

thread-count.c

```c
  // 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;
}
```

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 1000000 2
```

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

```
./count 1000000 16
```

Q6: What is going on???