These state diagrams assume that a process can only reach the "Terminated" state from the "Running" state. How could a process in the "Runnable" or "Waiting" state transition to the "Terminated" state?
User could kill process before the process has called exit()!
Ex: Control-C a process
Ex: kill -9 1831
User could kill process before the process has called exit()!

Ex: Control-C a process \(\rightarrow\) Generates SIGINT
Ex: kill -9 1831
User could kill process before the process has called exit()!

- Ex: Control-C a process → Generates SIGINT
- Ex: kill -9 1831 → Generates SIGKILL
Define a simple C function that, when invoked, will eventually cause a stack overflow. Then describe how the stack overflow might lead to data corruption of heap objects.

```c
unsigned int factorial(unsigned int n){
    if(n == 1){
        return 1;
    }else{
        return n * factorial(n-1);
    }
}

factorial(6); //Works as expected.
factorial(0); //Disaster strikes!
```
Define a simple C function that, when invoked, will eventually cause a stack overflow. Then describe how the stack overflow might lead to data corruption of heap objects.

```c
unsigned int factorial(unsigned int n){
    if(n == 1){
        return 1;
    }else{
        return n * factorial(n-1);
    }
}
```

```
factorial(6); //Works as expected.
factorial(0); //Disaster strikes!
```
Case study: Linux kernel

- **Per-cpu kernel stack**
  - (hardware interrupts)
  - 4 KB or 8 KB

- **Per-user-thread kernel stack**
  - (system calls, exceptions)
  - 4 KB or 8 KB

- Stack start (high addr)
- Stack end (low addr)
Case study: Linux kernel

Linux kernel stack overflow when mounting ISO9660 image: “We use a long chain of unique inode references (100+). Because the resolution of the chain is implemented via recursive functions, we explode the kernel stack.” [https://code.google.com/p/google-security-research/issues/detail?id=88](https://code.google.com/p/google-security-research/issues/detail?id=88)

Per-user-thread kernel stack (system calls, exceptions)

Stack start (high addr)

Stack end (low addr)

Per-cpu kernel stack (hardware interrupts)

4 KB or 8 KB

thread_info

4 KB or 8 KB
Case study: Linux kernel

Linux kernel stack overflow when mounting ISO9660 image: “We use a long chain of unique inode references (100+). Because the resolution of the chain is implemented via recursive functions, we explode the kernel stack.” [Link](https://code.google.com/p/google-security-research/issues/detail?id=88)

Per-user-thread kernel stack
(0x57AC6E9D)

4 KB or 8 KB

Per-cpu kernel stack
(hardware interrupts)

4 KB or 8 KB

Thread info

Stack start
(high addr)

Stack end
(low addr)