Assignment3

> *11911839 聂雨荷*

Q1

> ***[20 pts]*** *Read Chapter 2 of "Three Easy Pieces" (https://pages.cs.wisc.edu/~remzi/OSTEP/intro.pdf). Answer the following questions:*
>
> - *(1) What are the "three easy pieces" of operating systems? Explain each of them with your own words.*
> - *(2) How do these "three easy pieces" map to the chapters in the "dinosaur book"?*

- (1) The three easy pieces are:
    - **Virtualization**: OS maps the physical resources (memory, disk, CPU, etc.) into an easy-to-use virtual forms so that they can be easily used in the software level.
    - **Concurrency**: OS has to solve the problem of working with multiple things at the same time.
    - **Persistence**: OS coordinates the hardware and the software to enable the data being stored persistently.
- (2) In the "dinosaur book"
    - Chapters 3-5: process management -> virtualization
    - Chapters 6-8: process synchronization -> concurrency
    - Chapters 9-10: memory management -> virtualization
    - Chapters 11-12: storage management -> persistence
    - Chapters 13-15: file systems -> persistence
    - Chapters 16-17: security and protection -> virtualization

Q2

> ***[20 pts]*** *Read Chapter 6 of "Three Easy Pieces" (https://pages.cs.wisc.edu/~remzi/OSTEP/cpu-mechanisms.pdf) and explain what happens during context switch in detail?*

When an interrupt or an error happens, OS has regained control. OS will decide whether to continue running the current-running process or switch to a different one. If the decision is made to *switch*, then OS will execute a low-level piece of code, which is called the **context switch**.

[In a Nutshell] OS has to save a few values (general purpose registers, PC, kernel stack pointer, etc.) for the current-executing process and restore a few values for the soon-to-be-executing process.

When Process A is running and then is interrupted and OS decides to perform context switch. The following things will happen:

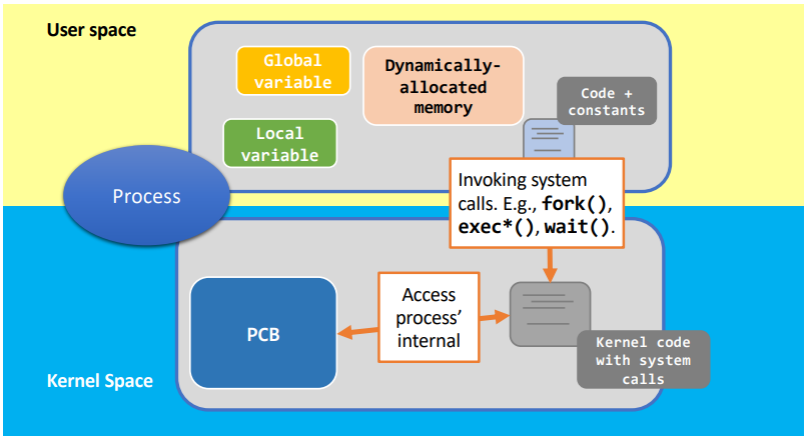
1. Save its registers(A) onto its kernel stack(A)
2. Switch to kernel mode
3. Jump to the trap handler
4. OS decides to switch from running process A to process B (context switch)
5. Call the `switch()` routine
    - Save current register values (A) into the process structure (A)
    - Restore the registers (B) from its process structure entry (B)
    - Switch context: Change the stack pointer to user B's kernel stack (and not A's)
6. OS returns-from-trap, restoring B's registers and starts running it

Q3

> ***[20 pts]*** *Read slides "L03 Processes I" and "L04 Processes II" and answer the following questions:*
>
> - *(1) Explain what happens when the **kernel** handles the fork() system call (hint: your answer should include the system call mechanism, PCB, address space, CPU scheduler, context switch, return values of the system call).*
> - *(2) Explain what happens when the kernel handles the exit() system call (hint: your answer should include discussion on the zombie state and how it is related to the wait() system call).*

- (1) When the kernel receives the `fork()` system call

a. The user code (User Mode) invokes the `fork()` system call, by executing a software interrupt

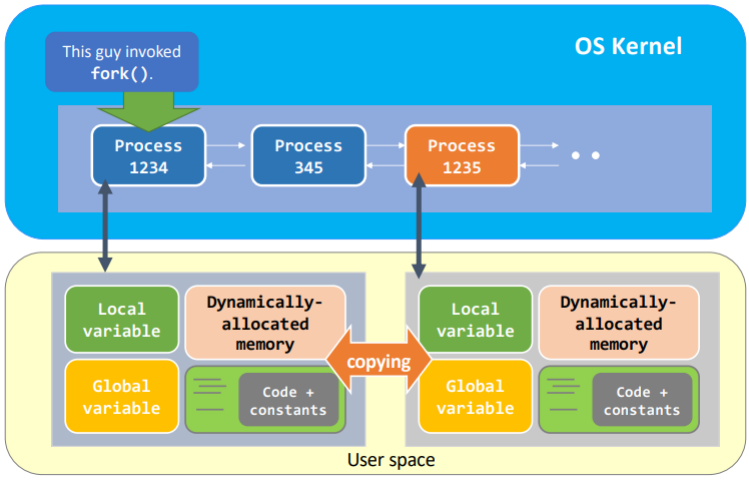


b. The kernel trap handler catches the interrupt, and transfers to the kernel mode

c. The kernel allocates a new process control block (PCB) for the child process. Also, it places the child process information at the end of the process linked list, waiting for the `schedule()` to perform context switch mentioned later.

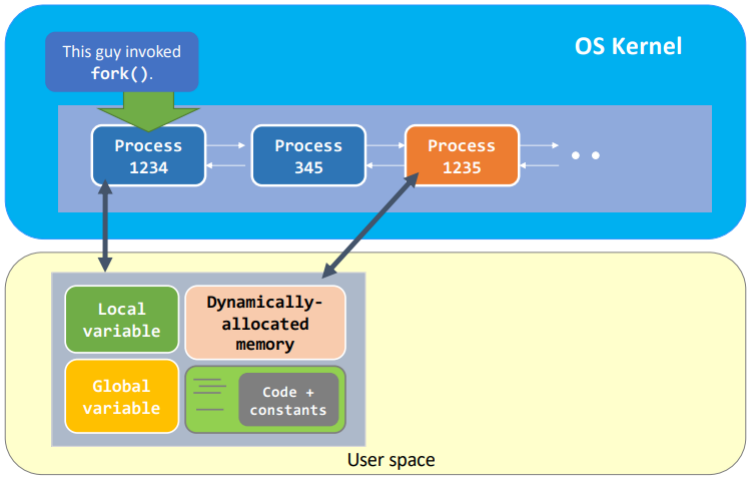
d. The kernel handles the copy of the address space

- duplicates: copying the contents of the parent process's address space into the corresponding locations in the child process's address space



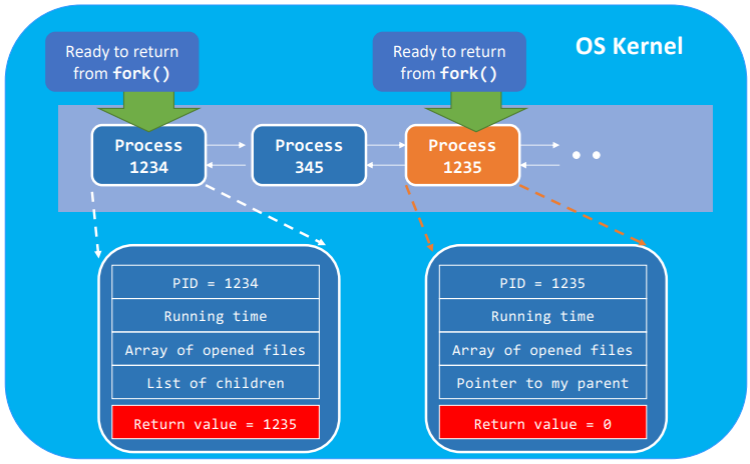
- copy-on-write: temporarily share the address space between the parent and the child process. When the data of the address

space is tend to be modified, then perform duplicates



e. The kernel initializes the child process's PCB with the appropriate values, including setting the child's process ID and parent process ID, allocating a new stack and program counter, and copying any relevant state from the parent process's PCB

f. The kernel sets the return value of the `fork()` system call. In the parent process, the return value is the process ID of the newly created child process, while in the child process, the return value is 0.

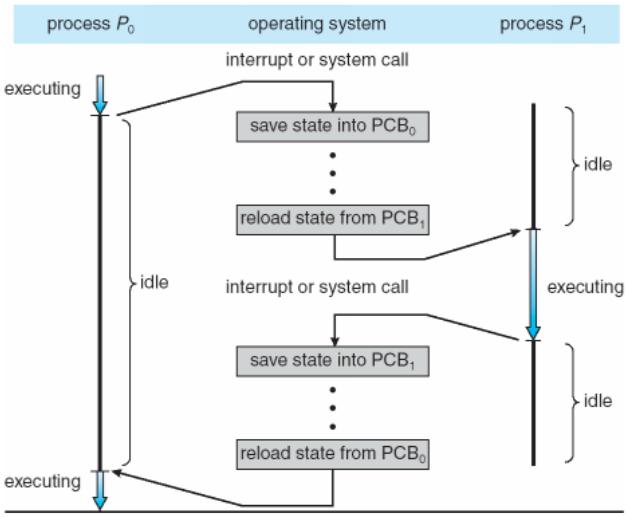


g. The system call returns control to the user mode

h. During later execution, the CPU scheduler then selects the next process to run, which may be either the parent or child process, depending on the scheduling algorithm used by the kernel

i. When the current process changes, a context switch occurs, which involves saving the current process's CPU state (registers, program counter, etc.) in its PCB and restoring the saved state for the newly selected process. (We have mentioned this process in the Question 2)

j. The selected process (either parent or child process) then begins executing from its saved program counter



k. Depends on the CPU's schedule mechanism, the parent and child processes execute independently

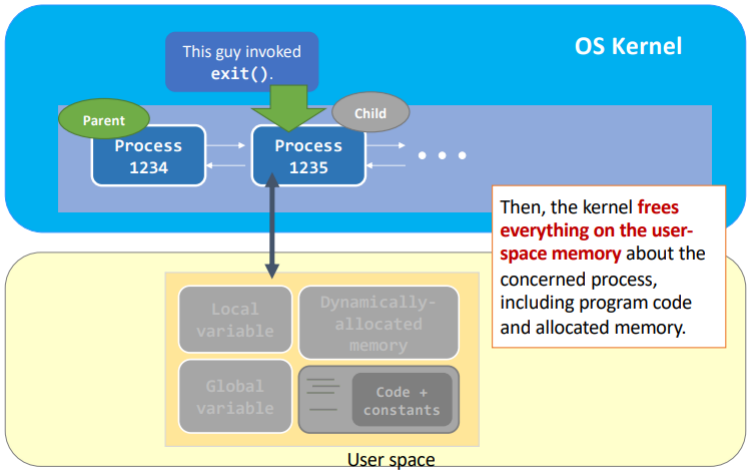
- (2) When the kernel receives `exit()` system call

a. The user code (User Mode) invokes the `exit()` system call, by executing a software interrupt

b. The kernel trap handler catches the interrupt, and transfers to the kernel mode

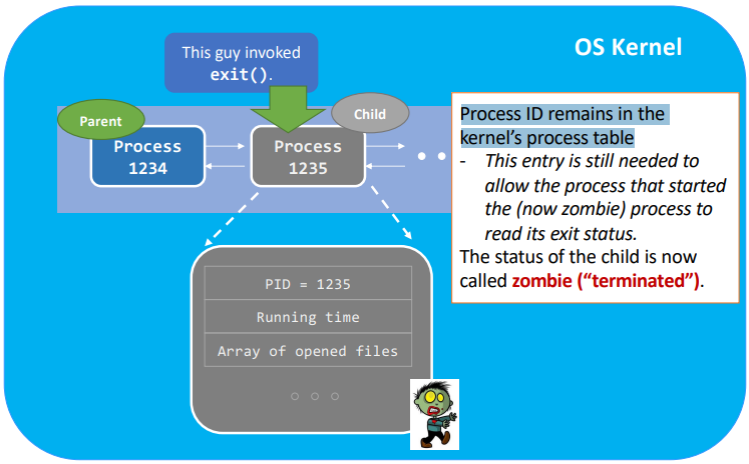
c. The kernel closes the list of opened files of this process

d. The kernel cleans the following things



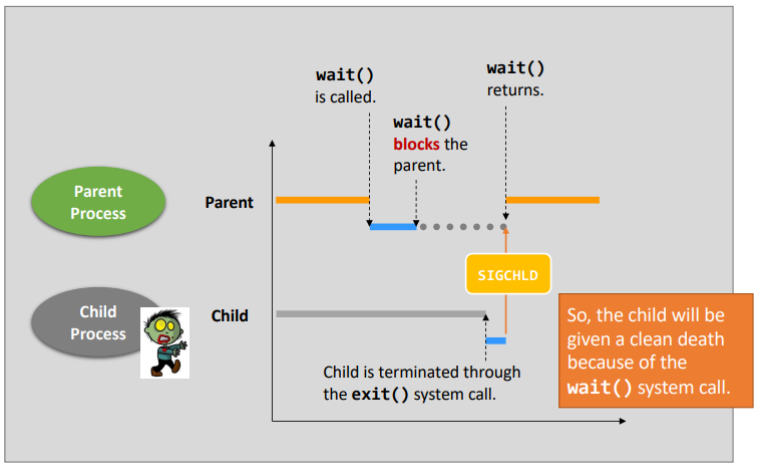
- Clean up most of the allocated kernel-space memory (e.g., process's running time info).
- Clean up the exit process's user-space memory, including program code and allocated memory

e. The kernel marks the process as `terminated` and updates the process state to the `zombie` state, which means that the process has completed its execution but has not yet been fully removed from the system. Process ID remains in the kernel's process table.

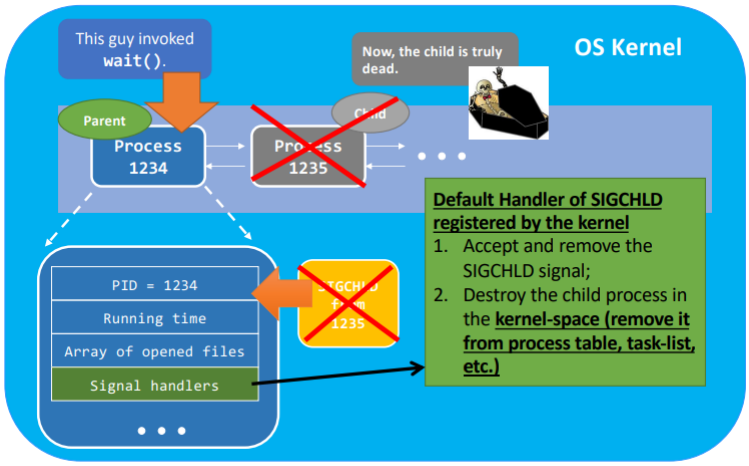


f. The kernel sends a `SIGCHLD` signal to the parent, notifying the parent the termination of its child.

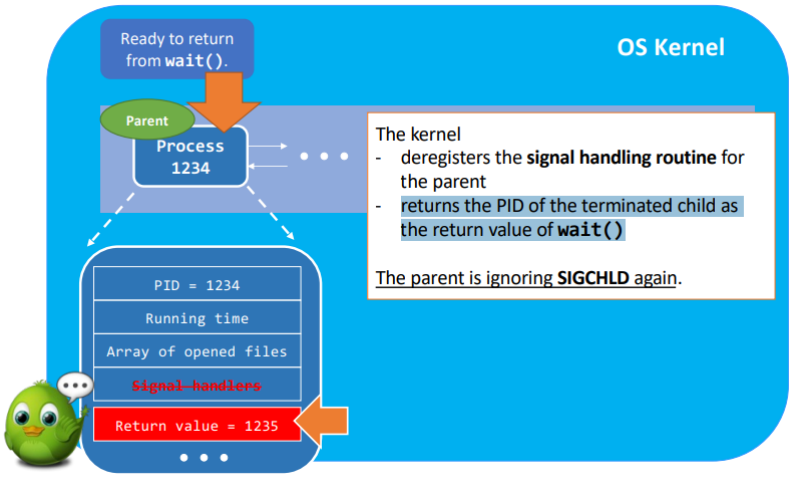
g. **[Normal Case]** The parent process can retrieve the exit status and resource usage information of its terminated child process by calling the `wait()` or `waitpid()` system call. These calls will block the parent process until one of its child processes terminates, at which point it will return the exit status and resource usage information of the terminated child process. If the parent process has registered to receive notification of its child's termination by calling the `wait()` system call, the kernel notifies the parent process, by `SIGCHLD` signal, that its child has terminated, and provides it with the exit status and resource usage information of the terminated process



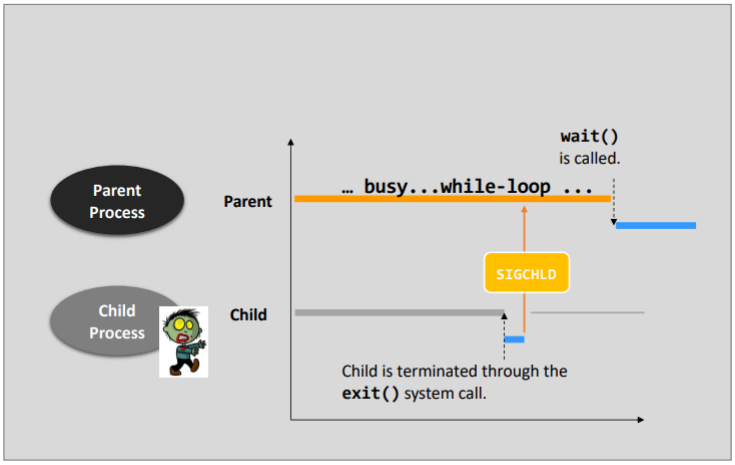
- The parent process invoke the signal handline routine



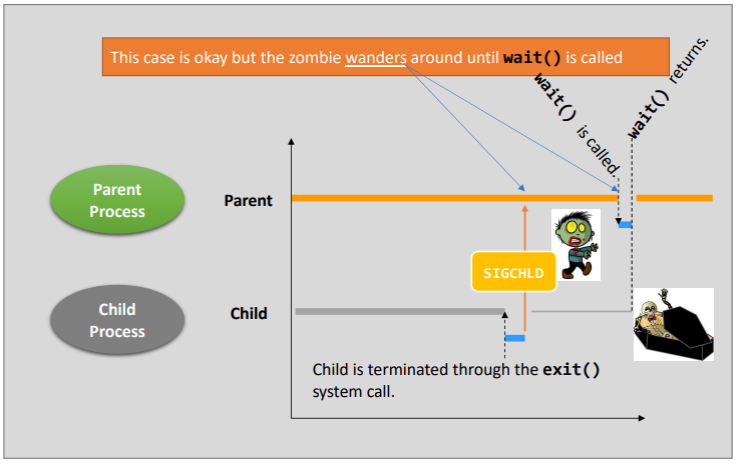
  - Accept and remove the `SIGCHLD` signal
  - Destroy the child process in the kernel-space (remove it from process table, task-list, etc.)
- The kernel deregisters the signal handling routine for the parent and returns the PID of the terminated child as the return value of `wait()`



h. **[Abnormal Case]** If the parent process has not registered to receive notification (invokes `wait()` or `waitpid()`)of its child's termination, the kernel will keep the zombie process in the system until the system is rebooted or the parent process terminates



- Before parent process invokes `wait()`, the `SIGCHLD` is already there. Thus it takes action immediately and returns.



Q4

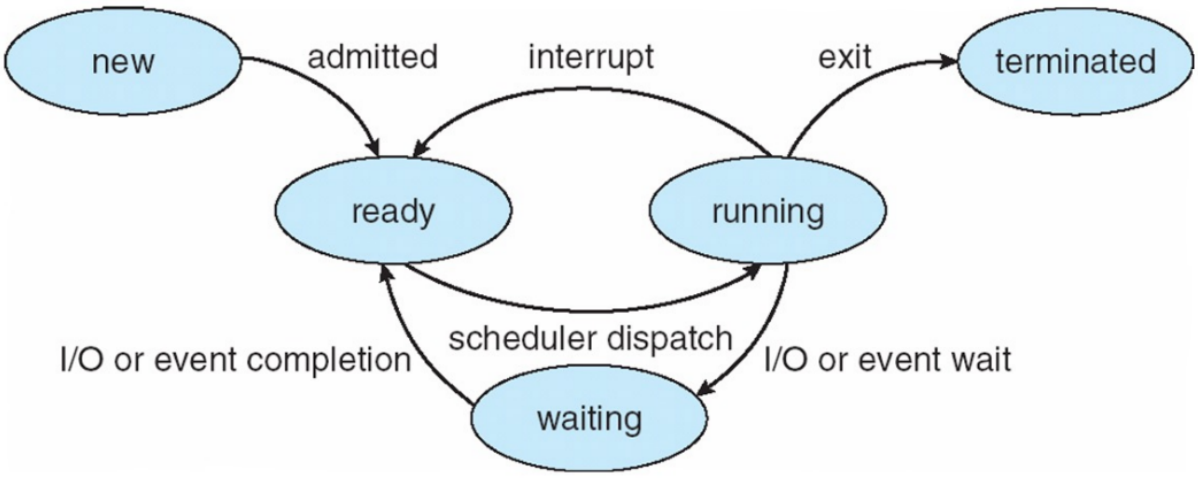
> ***[20 pts]*** *What are the three methods of transferring the control of the CPU from a user process to OS kernel? Compare them in detail.*

TYPES	DESCRIPTIONS	DIFFERENCES
System call	A system call is a specific method for a user process to request a service from the operating system It provides a standardized interface for user processes to interact with the kernel The user mode invokes the system call by executing a special instruction The processor then switches to the kernel mode and transfers control to the system call handler in the kernel The system call handler performs the requested service and returns control to the user process by executing a return-from-system-call instruction It provides a secure and efficient interface for user processes to access the kernel's services, and it is portable across different architectures and operating systems	Safety Calling
Interrupt	A software interrupt is a signal generated by a program or a process to request the kernel's service When the interrupt occurs, the processor automatically switches from user mode to kernel mode and transfers control to the interrupt handler The interrupt handler then performs the requested service and returns control to the user process by executing a return-from-interrupt instruction	Asynchronous Abnormal Condition

TYPES	DESCRIPTIONS	DIFFERENCES
Trap or Exception	An exception is an event that occurs during the execution of a program, such as a division-by-zero error or an invalid memory access When an exception occurs, the processor automatically transfers control to the exception handler in the kernel mode The exception handler then performs the necessary action, such as terminating the process or allocating memory, and returns control to the user process This method does not require the user process to explicitly call the interrupt instruction, but it requires the kernel to handle various types of exceptions, which may cause complexity and overhead	Synchronous Abnormal Condition

Q5

> ***[20 pts]*** *Describe the life cycle of a process (hint: explain the reasons for process state transitions)*



STATES	DESCRIPTIONS
New	A process is in the new state when it is first created. The process is allocated system resources, such as memory and CPU time, but has not yet been admitted to the set of running processes. In this state, the process is waiting to be loaded into main memory.

STATES	DESCRIPTIONS
Ready	When a process has been loaded into main memory and is awaiting CPU time, it is said to be in the ready state. The process has been added to the process linked list. In this state, the process is eligible for execution, but the CPU scheduler has not yet selected it for execution.
Running	A process is said to be in the running state when the CPU is executing its instructions. Only one process can be in the running state on a single processor at a time.
Waiting	When a process is waiting for some event, such as an I/O operation or a signal, it is said to be in the waiting state. In this state, the process is not eligible for execution, as it cannot make any progress until the event it is waiting for occurs.
Terminate	A process enters the terminated state when it has finished executing and is no longer needed. The operating system may reclaim any resources allocated to the process, such as memory or file handles, at this time.

TRANSITIONS	DESCRIPTIONS
Admitted (New -> Ready)	The process finishes all its preparation and starts waiting for the scheduling of the CPU.
Scheduler Dispatch (Ready -> Running)	CPU selects a process from the ready queue and CPU resources to that process. This involves updating the process control block (PCB), sets the CPU registers and program counter. The selected process is then dispatched and begins executing its instructions on the CPU.
Interrupt (Running -> Ready)	The process is interrupted by other processes. For example, the time interrupt. It will stop executing, record the require value into PCB, save all the current data. The process will be re-inserted to the ready queue and wait for the next schedule.
I/O or event wait (Ready -> Wait)	If a process is waiting for a resource to become available, such as a network connection, user input or file reading, it may enter a waiting state until that resource is ready.
I/O or event completion (Wait -> Ready)	When the require resource is ready. The process will re-enter to the Ready state and be added to the ready queue, waiting for the next schedule.
Exit (Running -> Terminate)	When the process finishes the whole execution instructions and is no longer needed. It will transform from the Running state to the Terminate state.