Process management

> Program: the written code (static)
>
> Process: a program that has started to be executed (dynamic)
>
> - A program which has been placed in the memory
> - with a process control block created
> - or has at least once entered a running state
>
> **Process control block(PCB)**: a data structure which contains all of the data needed for a process to run; this can be created in memory when data needs to be received during execution time. The PCB will store
>
> - **current process state** (ready, running, or blocked)
> - process privileges(such as which resources is allowed to access)
> - register values (PC, MAR, MDR, and ACC)
> - process priority and any **scheduling information**
> - the amount of CPU time the process will need to complete
> - a **process ID** which allows it to be uniquely identified
> - **I/O status information** - I/O devices allocated to process, list of open files
>
> **Process context** - the machine environment while the process is running
>
> - **Program counter**, location of instruction to next execute
> - **CPU registers**, contents of all process-centric registers
>
> **Multitasking**: allow computers to carry out more than one task(process) at a time. Each of these processes will share common hardware resources. To ensure multitasking operates correctly, scheduling is used to decide which process is carried out first.

Process states

Description

- New: process is being created
- Ready: program waiting for CPU time

    - The process is not being executed
    - The process is in the queue
    - … waiting for the processor’s attention/time slice/CPU time
- Waiting/Blocked: program waiting for an event or Input/Output operation
    - The process is waiting for an event
    - So it cannot be executed by the moment
    - e.g. Input/Output
- Running: program running on a CPU
    - Process is being executed by the CPU
    - Process is currently using its allocated time slice
- Terminated(Exit): process has finished executing

Transition between states: conditions

- Ready -> Running
- Current process no longer running // processor is available
    - Process was at the head of the ready queue // process has the highest priority
- Blocked -> Ready
- The only required resource become available // resource/event/IO operation is complete
- Running -> Ready
    - When process is executing it is allocated a time slice
    - When the time slice is completed, the process can no longer use the processor even if it is capable of further processing
- Running -> Blocked
- Process is executed when it needs to perform IO operation
    - Placed in blocked state until the IO operation is complete

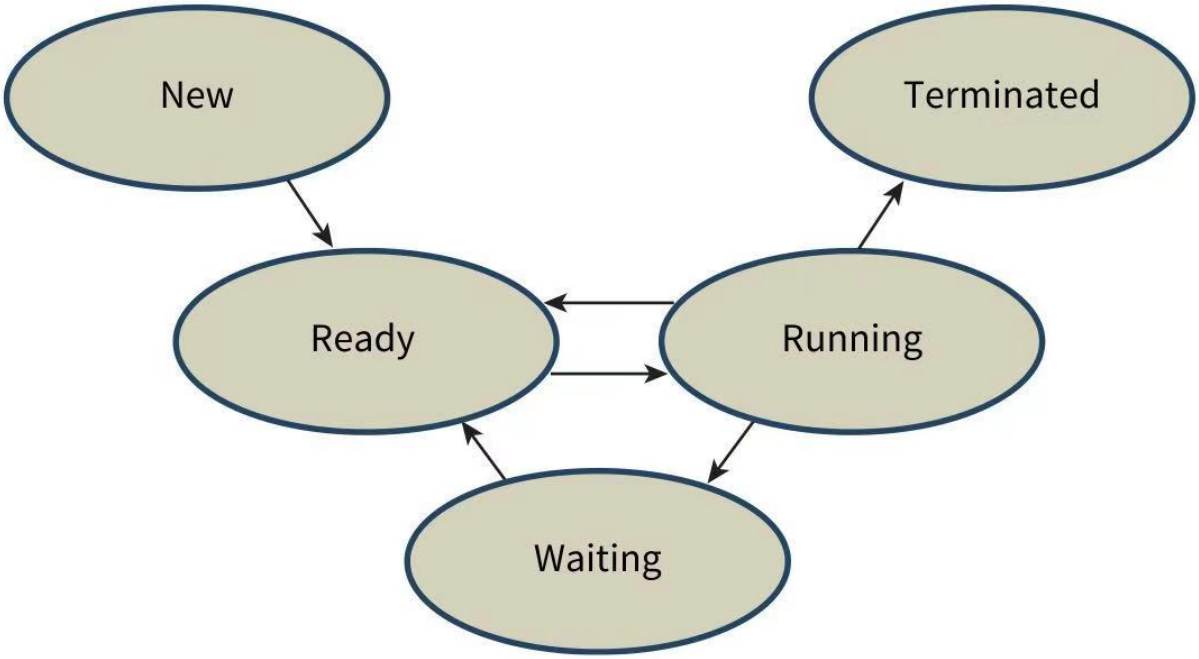

Figure 20.05 The five states defined for a process being executed

The transition between the states shown in Figure 20.05 can be described as follows

- A new process arrives in memory and a PCB is created; it changes to the ready state.
- A process in the ready state is given access to the CPU by the dispatcher; it changes to the running state
- A process in the running state is halted by an interrupt; it returns to the ready state
- A process in the running state cannot progress until some event has occurred(IO perhaps); it changes to the waiting/blocked state.
- A process in the waiting state is notified that an event is complete; it returns to the ready state.
- A process in the running state completes execution; it changes to the terminated state

---

Explain how the processes are affected when the following event takes place

- The running process needs to read a file from a disk
  - Running process is halted
  - Process moved to blocked state
- The running process using up its time slice

- Running process is halted // another process has the use of the processor
- Process moves to ready state
- … until next time slice is allocated

Explain why a process cannot be moved from the blocked state to the running state

- When IO operation is completed for the process in blocked state
- Process put into ready queue
- OS determine which process get next use of the processor from the ready queue

Explain why a process cannot be moved from the ready state to the blocked state

- To be in blocked state process must initiate some IO operation
- To initiate operation, process must be executing
- If process in ready state, cannot be executing; hence must in running state

Process scheduling

> **High-level scheduler** controls the selection of a program stored on disk to be moved into main memory.
>
> - Decides which processes are loaded from backing store
> - Into memory/ready queue
>
> **Low-level scheduler** controls when the program installed in the memory has access to the CPU
>
> - Decide which process should next get the use of CPU time
> - Based on position/priority
> - Invoked after interrupt/OS call
> - Purpose: ensure response time is acceptable and the system remains stable at all times.

Although the high-level and low-level scheduler will have decisions to make when choosing which program should be loaded into memory, we concentrate here on the options for the **low-level scheduler**.

Round Robin

Shortest job first

First come first serve

Shortest remaining time

Explain why the operating system needs to use scheduling algorithm

- To allow multitasking to take place
- To ensure fair usage of the processor, peripheral, and memory
- To ensure high priority tasks are executed sooner
- To ensure all processes have the opportunity to finish
- To keep CPU busy all the time
- To service largest amount of job in a given amount of time
- To minimize the time user must wait for their result

Kernel

> Central component responsible for communication between hardware, software, and memory
>
> Responsible for:
>
> - Process management
> - Device management
> - Memory management
> - Interrupt handling
> - Input/output file communication

Interrupt handling & OS kernel

> Interrupt: signal from a hardware or software device, seeking the attention of the processor

1. The CPU will check for interrupt signals. The system will enter the kernel mode if any of the following type of interrupt signals are sent
    - Device interrupt (printer out of paper, device not present)
    - Exceptions (instruction faults: division by zero, unidentified opcode, stack faults)
    - Traps/software interrupt (process requesting a resource such as disk drive)
2. Kernel will consult the interrupt dispatch table(IDT) - links a device description with the proper interrupt routine
3. IDT will supply the address of the low-level routine to handle the interrupt event occurred
4. The kernel will save the state of the interrupt process on the kernel stack
5. The process state will be restored once the interrupting task is services.

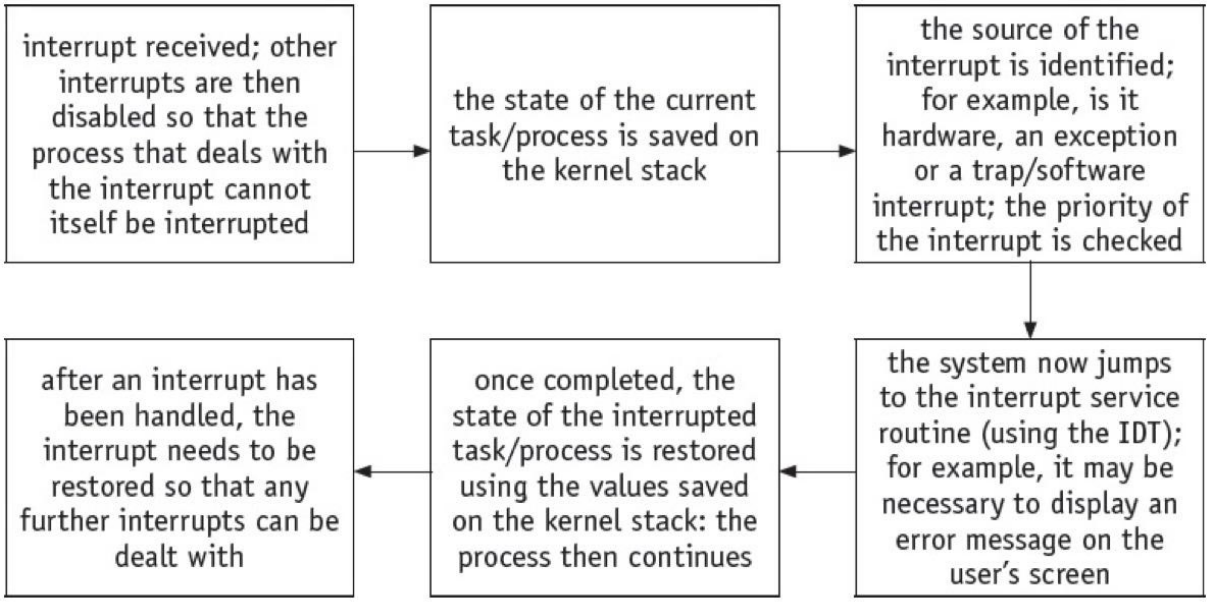

Figure 16.9

Describe the sequence of steps which would be carried out by the interrupt handler software when an interrupt is received and serviced

- Identify the source of interrupt
- Disable all interrupts of low priority
- Save the content of the PC
- Save the content of other registers on the kernel stack
- Load and run the appropriate interrupt service routine(ISR)
- Restore registers from the stack
- Enable all interrupts

- Continue execution of the interrupts