Processes and Threads

Week 04 – Lecture 1
Processes

Team

Instructors

Giancarlo Succi

Joseph Brown

Teaching Assistants

Vladimir Ivanov

Stanislav Litvinov

Alexey Reznik

Munir Makhmutov

Hamna Aslam

Sources

- These slides have been adapted from the original slides of the adopted book:
  - Tanenbaum & Bo, Modern Operating Systems: 4th edition, 2013
    Prentice-Hall, Inc.

  and customised for the needs of this course.
- Additional input for the slides are detailed later

Processes (1)

- A **process** is an abstraction of a running program
- Processes support the ability to have (pseudo) concurrent operation even when there is only one CPU available
- Examples:
  - Web-server serves many requests from different users simultaneously
  - OS on User PC is able to run several processes: email-client, antivirus, word processor, and so on

Processes (2)

- In any multiprogramming system, the CPU switches from process to process quickly
- At any one instant the CPU is running only one process
- In the short period of time it may work on several of them processes
- Such an illusion of parallelism is called **pseudo-parallelism** to contrast it with the **true hardware parallelism** of systems that have two or more CPUs (cores)
- OS designers over the years have evolved a conceptual model (sequential processes) that makes parallelism easier to deal with

The Process Model (1)

- All the runnable software on the computer, sometimes including the OS, is organized into a number of **(sequential) processes**
- A process is an instance of an **executing program** including the current values of the program counter, registers, and variables
- **Conceptually**, each process has its own virtual CPU
- In reality, the real CPU switches between different processes. Such a switching is called **multiprogramming**

The Process Model (2)

- There is only one physical program counter, so when each process runs, its logical program counter is loaded into the real program counter. When it is finished (for the time being), the physical program counter is saved in the process' stored logical program counter in memory

The Process Model (3)

- Let's consider an example of a computer multiprogramming four programs in memory

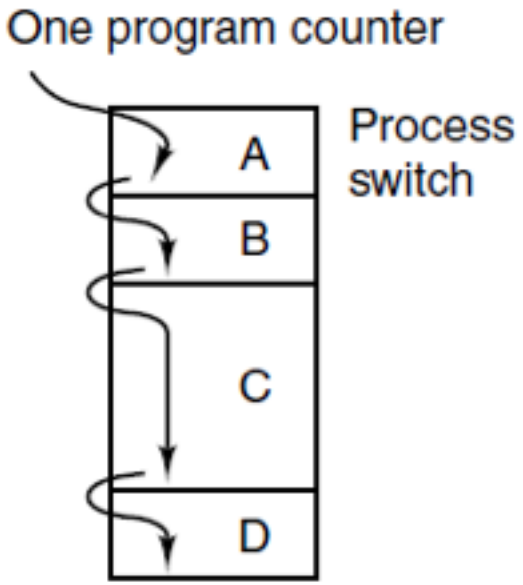


Figure 2-1. (a) Multiprogramming of four programs.

The Process Model (4)

- Each of the processes has its own flow of control (i.e., its own logical program counter) and runs independently of the other ones

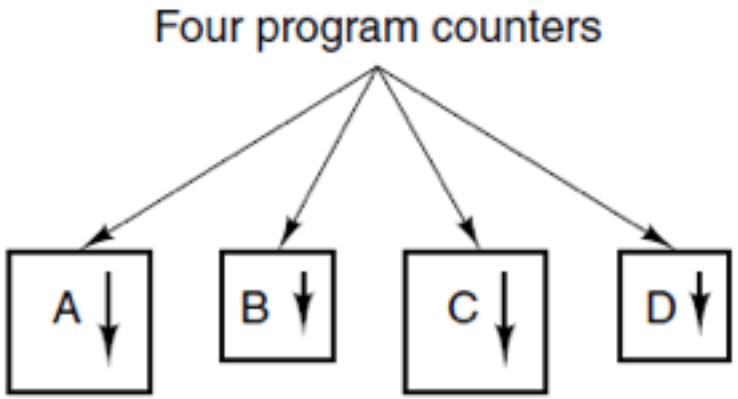


Figure 2-1. (b) Conceptual model of four independent, sequential processes.

The Process Model (5)

- All the processes have made progress, but at any given instant only one process is actually running

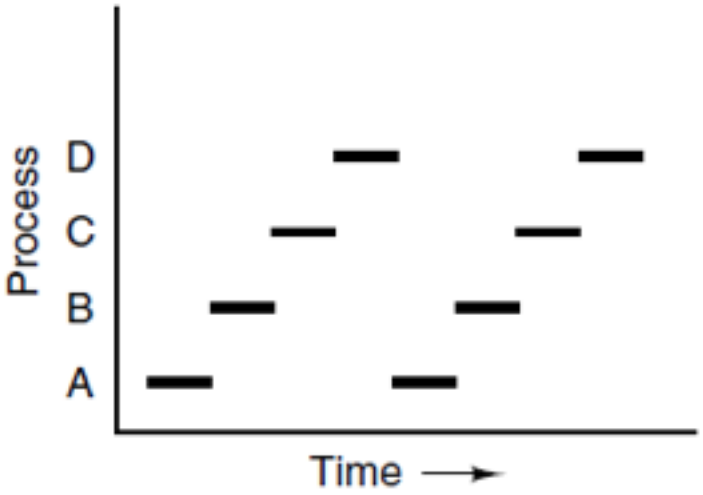


Figure 2-1. (c) Only one program is active at once.

The Process Model (6)

- The difference between a program and a process:
  - A program is a set of instructions. It is not doing anything (like a recipe from recipe book)
  - A process is is an activity of some kind. It has a program, input, output, and a state (like a cook following a recipe from a book)
- If a program is running twice, it counts as two distinct processes

Process Creation (1)

- Four principal events that cause processes to be created:
  - System initialization
  - Execution of a process creation system call by a running process
  - A user request to create a new process
  - Initiation of a batch job

Process Creation (2)

- System initialization:
  - When an operating system is booted, typically numerous processes are created
  - Some of these are foreground processes that interact with users and perform work for them
  - Others run in the background and have some specific function
  - Processes that stay in the background to handle some activity such as email, Web pages, news, printing, and so on are called **daemons**

Process Creation (3)

- Execution of a process creation system call by a running process:
  - A running process can issue system calls to create one or more new processes to help it do its job
  - For example, if a large amount of data is being fetched over a network for subsequent processing, one process may be created to fetch the data and put them in a shared buffer while a second process may remove the data items and process them
  - If the system has multiple CPUs, it might help to perform the job faster

Process Creation (4)

- A user request to create a new process:
  - A simple situation in interactive systems when a user types a command in shell or clicks an icon of an application
  - It starts a new process and runs the selected program in it
  - In command-based UNIX systems the new process takes over the window in which it was started
  - In Windows process usually creates a new window
  - In both systems, users may have multiple windows open at once, each running some process

Process Creation (5)

- Initiation of a batch job:
  - applies only to the batch systems found on large mainframes
  - users can submit batch jobs to the system (possibly remotely)
  - When the OS decides that it has the resources to run another job, it creates a new process and runs the next job from the input queue in it

Process Creation (6)

- A new process is created by having an existing process execute a process creation system call that tells the OS to create a new process and indicates which program to run in it
- In UNIX, there is only one system call to create a new process: **fork**, which creates an exact clone of the calling process
- After the fork, the parent and the child processes, have the same memory image, environment strings and open files
- Usually, the child process then executes **execve** or a similar system call to change its memory image and run a new program
- The child can manipulate its file descriptors after the **fork** but before the **execve** in order to accomplish redirection of standard input, standard output, and standard error

Process Creation (7)

- In Windows a single Win32 function call, **CreateProcess**, handles both process creation and loading the correct program into the new process
- This call has 10 parameters, including:
  - the program to be executed
  - the command-line parameters to feed that program
  - various security attributes
  - bits that control whether open files are inherited
  - priority information
  - a specification of the window to be created for the process (if any)
  - a pointer to a structure in which information about the newly created process is returned to the caller.

Process Creation (8)

- In both UNIX and Windows systems, after a process is created, the parent and child have their own distinct address spaces.
- In UNIX, the child's initial address space is a copy of the parent's, but there are definitely two distinct address spaces involved; no writable memory is shared
- In Windows, the parent's and child's address spaces are different from the start
- The child in UNIX may the parent's memory, but in that case the memory is shared **copy-on-write**:
  - chunk of memory is explicitly copied before writing to it to make sure the modification occurs in a private memory area
- It is possible for a newly created process to share some of its creator's other resources, such as open files

Process Termination (1)

- Typical conditions which terminate a process:
  - Normal exit (voluntary)
  - Error exit (voluntary)
  - Fatal error (involuntary)
  - Killed by another process (involuntary)

Process Termination (2)

- Most of the processes terminate because they are done with their jobs
- To do this, they execute a special system call (exit in UNIX or ExitProcess in Windows)
- A process might voluntarily terminate when it faces an error condition caused by some external sources (input parameters). In this case a process will usually record an error and exit

Process Termination (3)

- A process might terminate because of a fatal error
- Examples include:
  - executing an illegal instruction
  - referencing nonexistent memory
  - dividing by zero
- In some systems (e.g., UNIX), a process can tell the OS that it wishes to handle certain errors itself, in which case the process is signaled (interrupted) instead of terminated
- A process might terminate when it executes a system call telling the OS to kill some other process. In UNIX this call is **kill**. The corresponding Win32 function is **TerminateProcess**

Process Hierarchies (1)

- In some systems the parent process and child process continue to be associated in certain ways. The child process can itself create more processes, forming a **process hierarchy**
- In UNIX, a process and all of its children and further descendants together form a **process group**
- For example, when a user sends a signal from the keyboard, the signal is delivered to all members of the process group currently associated with the keyboard. Each process can catch the signal, ignore the signal, or to be killed by the signal (default action)

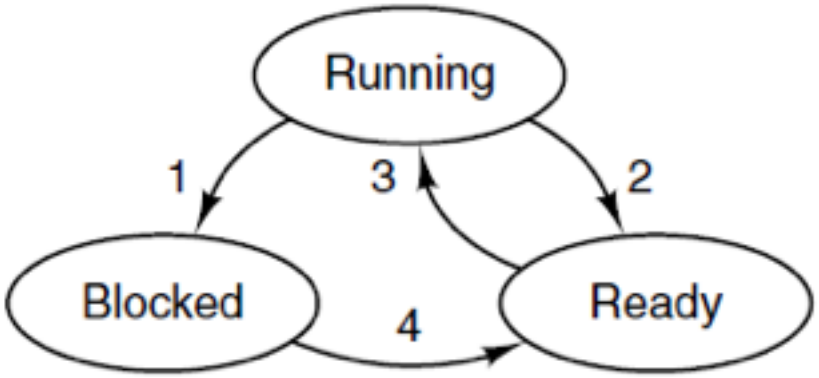
Process Hierarchies (2)

- Another example:
  - when UNIX is started, a process called **init** reads a file telling how many terminals there are and forks off a new process per terminal
  - these processes wait for someone to log in. If a login is successful, the login process executes a shell to accept commands that may start up more processes
  - all the processes **in the whole system** belong to a single tree, with **init** at the root
- Windows has no concept of a process hierarchy, all processes are equal. When a process is created, the parent is given a special token (called a handle) that it can use to control the child
- It is free to pass this token to some other process, thus invalidating the hierarchy
- In contrast, processes cannot disinherit their children in UNIX

Process States (1)

- Three states a process may be in (Fig. 2-2):
  - **Running** (actually using the CPU at that instant)
  - **Ready** (runnable; temporarily stopped to let another process run)
  - **Blocked** (unable to run until some external event happens)
- When a process blocks, it does so because logically it cannot continue (waiting for input that is not yet available)
- Different situation is when OS decides to allocate the CPU to another process for a while

Process States (2)



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Figure 2-2. A process can be in running, blocked, or ready state. Transitions between these states are as shown.

Process States (3)

- Transaction 1 (Running → Blocked):
  - occurs when the OS discovers that a process cannot continue right now
  - in some systems the process executes a system call, such as **pause**
  - in UNIX and some other systems the process is blocked automatically
- Transaction 2 (Running → Ready):
  - occurs when the scheduler decides that it is time to let another process have some CPU time

Process States (4)

- Transaction 3 (Ready → Running):
  - occurs when it is time for the first process to get the CPU to run again since all the other processes have run long enough
- Transaction 4 (Blocked → Ready):
  - occurs when the external event for which a process was waiting (such as the arrival of some input) happens
  - if no other process is running at that moment, transition 3 will be triggered and the process will start running

Process States (5)

- Using the process model, it becomes much easier to think about what is going on inside the system
- Instead of thinking about interrupts, we can think about user processes, disk processes, terminal processes, and so on, which block when they are waiting for something to happen
- When the disk has been read or the character typed, the process waiting for it is unblocked and is eligible to run again
- In this model the lowest level of the OS is the scheduler (Fig. 2-3) with a variety of processes on top of it. However, few real systems are as nicely structured as this

Process States (6)

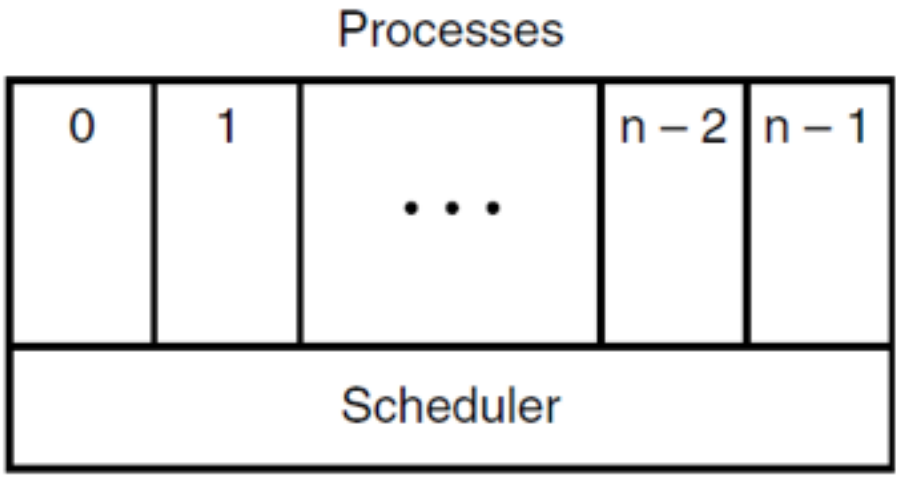
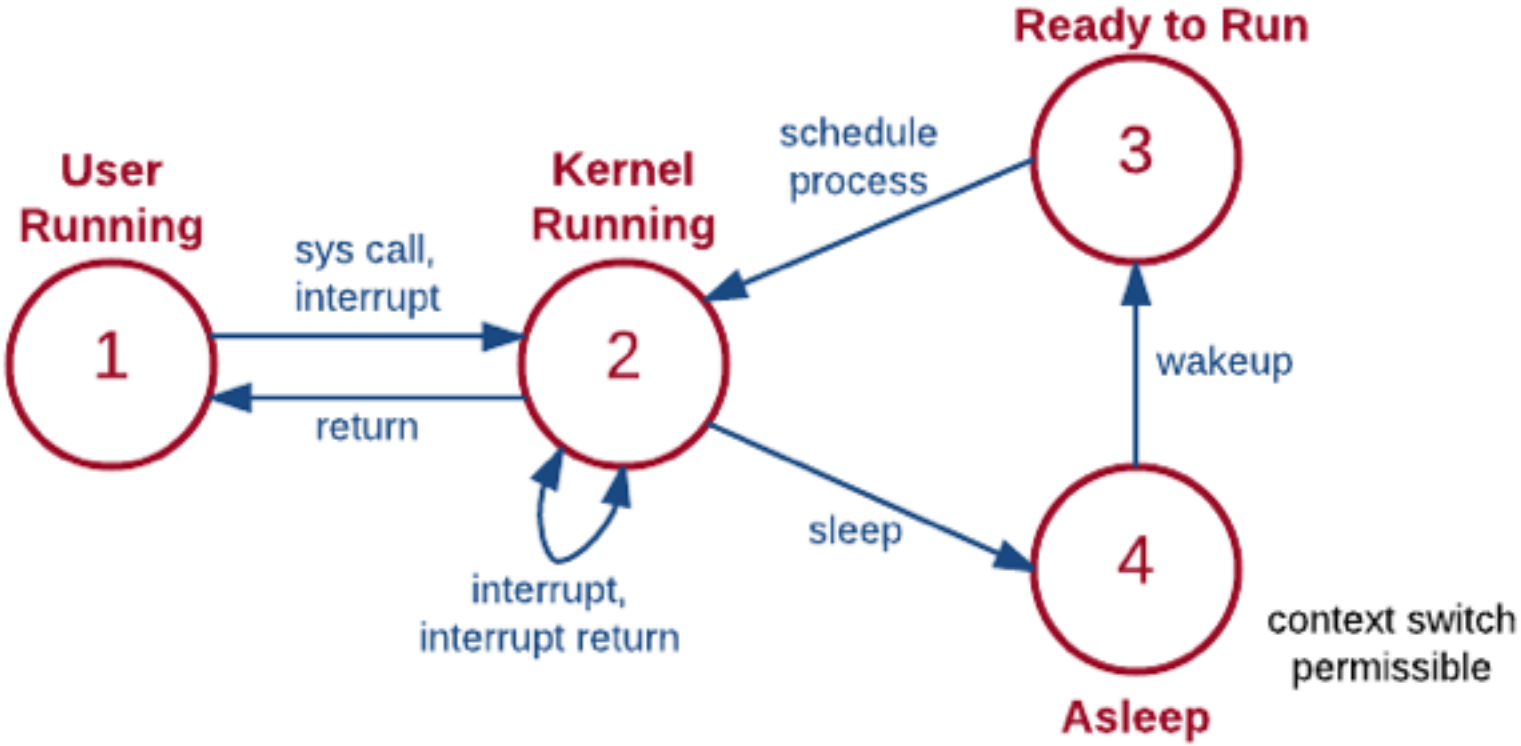


Figure 2-3. The lowest layer of a process-structured operating system handles interrupts and scheduling. Above that layer are sequential processes.

Process States (7)

- The deeper view on process states is the following:
  1. The process is currently executing in user mode
  2. The process is currently executing in kernel mode
  3. The process is not executing, but it is ready to run as soon as the scheduler chooses it. Many processes may be in this state, and the scheduling algorithm determines which one will execute next
  4. The process is *sleeping*. A process puts itself to sleep when it can no longer continue executing, such as when it is waiting for I/O to complete

Process States (8)



Process States and Transitions

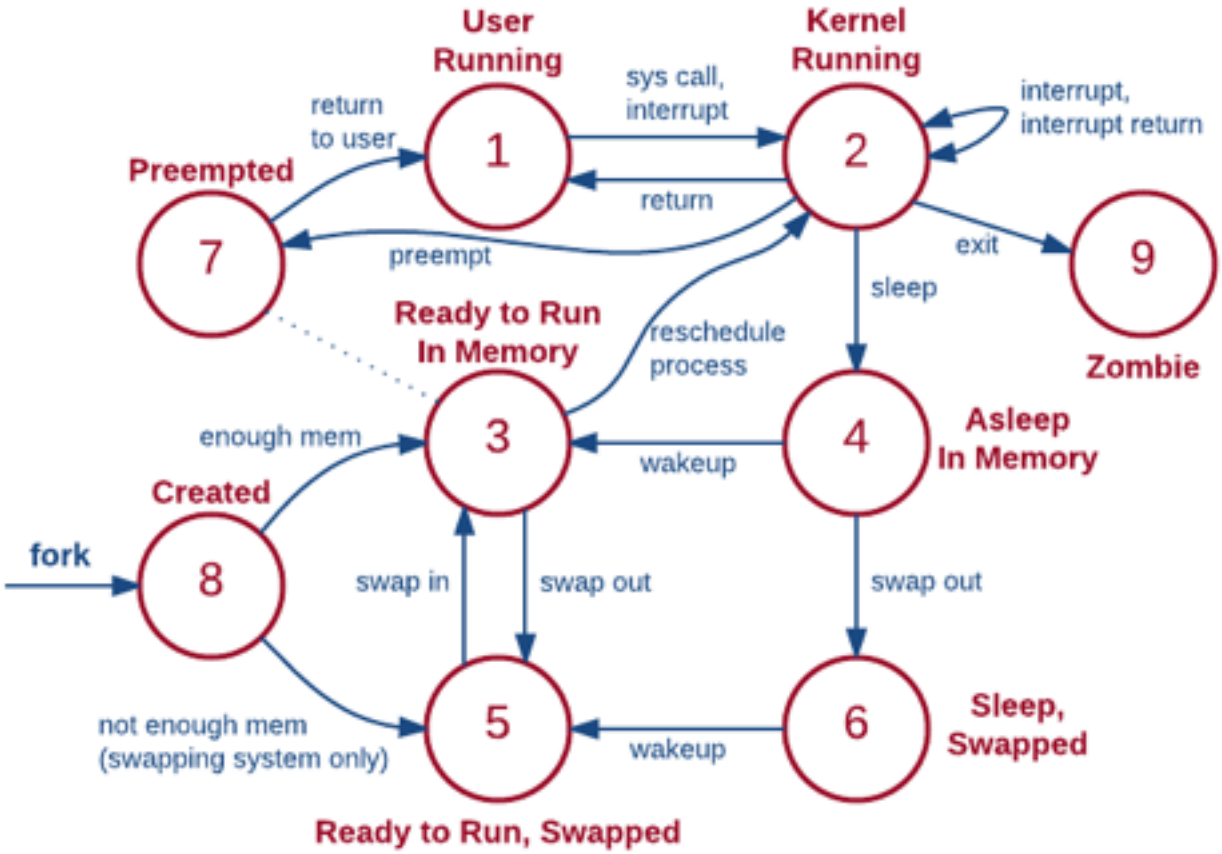
Process States (9)

- The complete list of process states is:
  1. The process is executing in user mode
  2. The process is executing in kernel mode
  3. The process is not executing but is ready to run as soon as the kernel schedules it
  4. The process is sleeping and resides in main memory
  5. The process is ready to run, bit the swapper (process 0) must swap the process into main memory before the kernel can schedule it to execute

Process States (10)

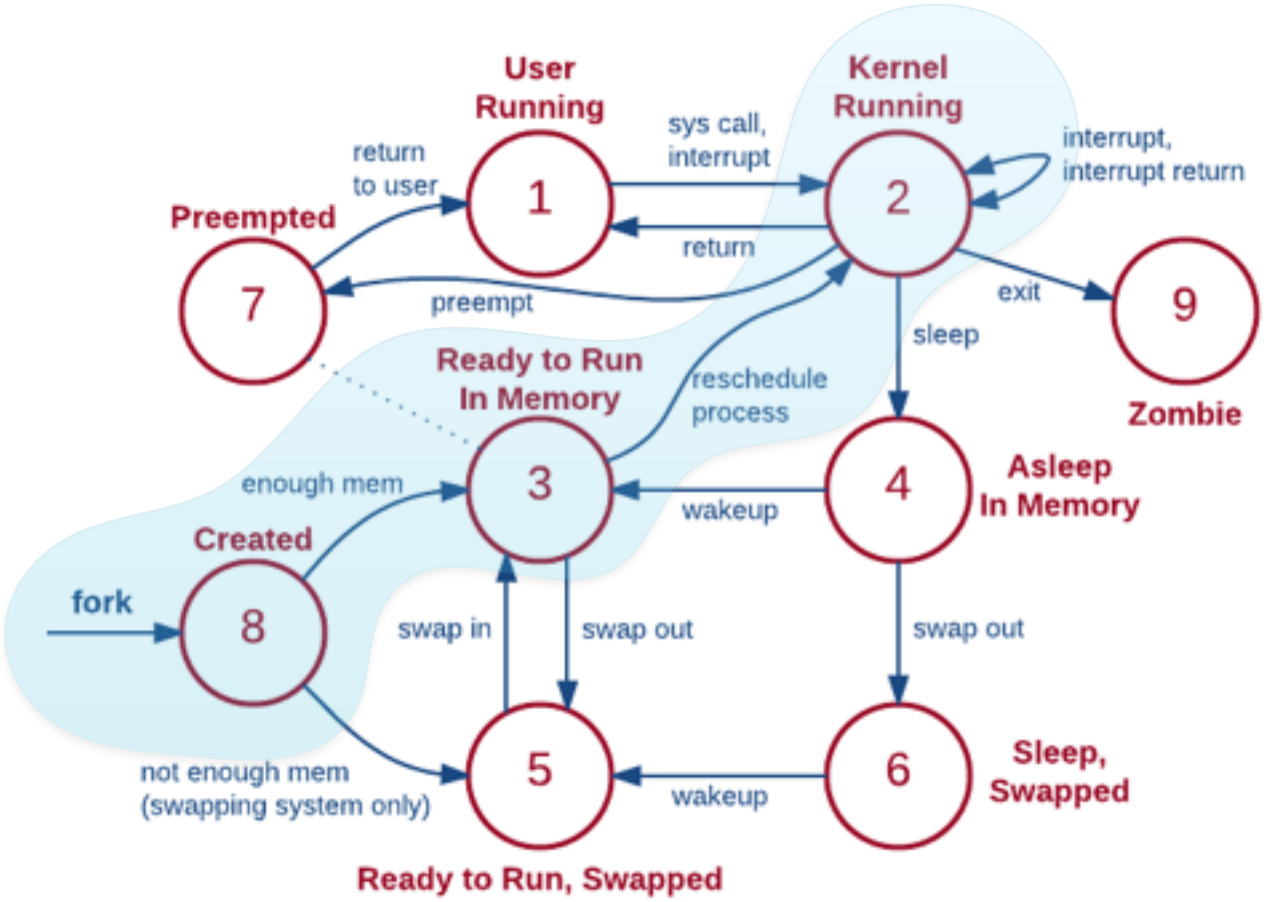
6. The process is awaiting an event and has been swapped to secondary storage (a blocked state)
7. The process is returning from kernel to user mode, but the kernel preempts it and does a context switch to schedule another process
8. The process is newly created and is in a transition state; it is not yet ready to run, nor it is sleeping (the initial state for all the processes except process 0)
9. Process executed the exit system call and no longer exists, but it leaves a record for its parent process to collect (the final state of a process)

Process States (11)



Process State Transition Diagram

Process States (12)

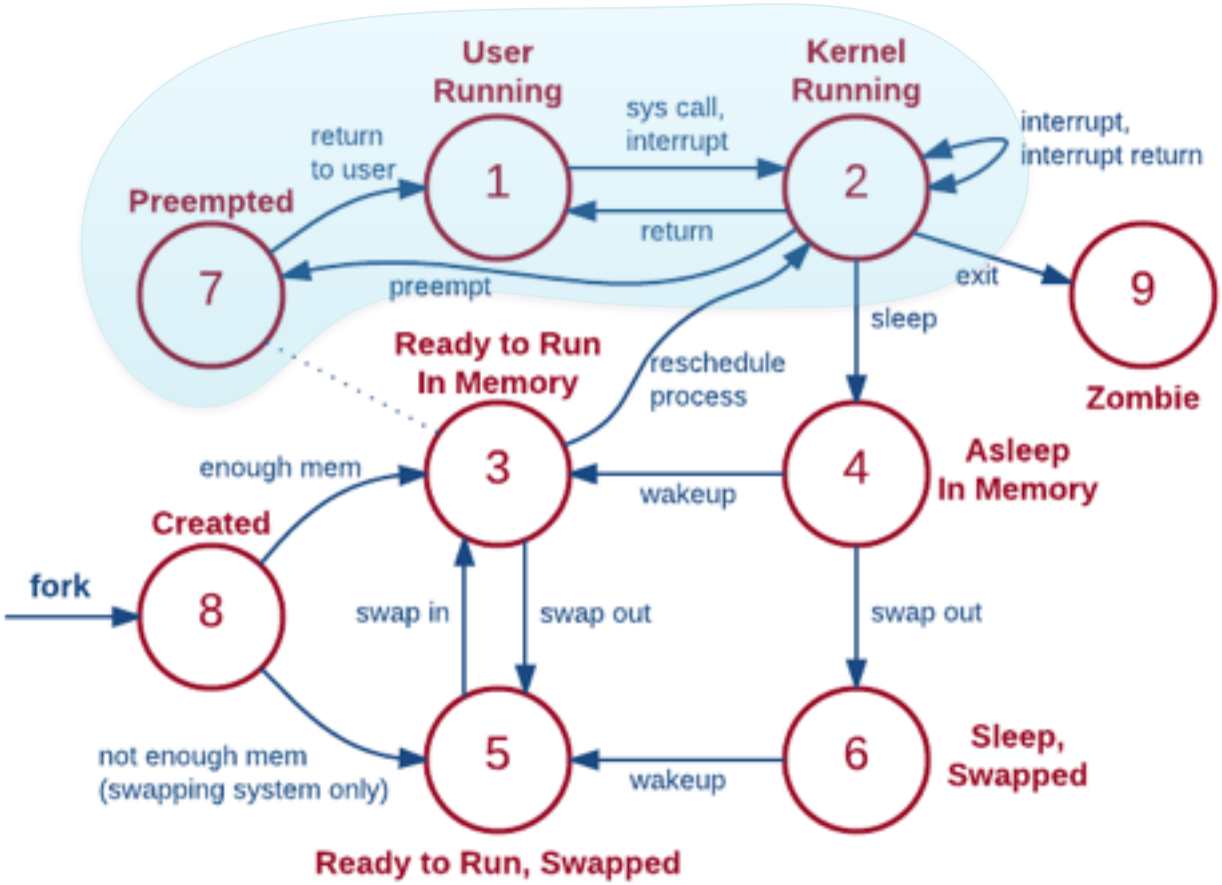


Process State Transition Diagram

Process States (13)

- Consider a typical process as it moves through the state transition model:
  - The process enters the state model in the "created" state (after the parent process executes *fork* system call). Eventually, it moves to one of the "ready to run" states (3 or 5)
  - Let's assume it enters the state "ready to run in memory"
  - The process scheduler will eventually pick the process which will enter the state "kernel running" and will complete its part of the *fork* system call

Process States (14)

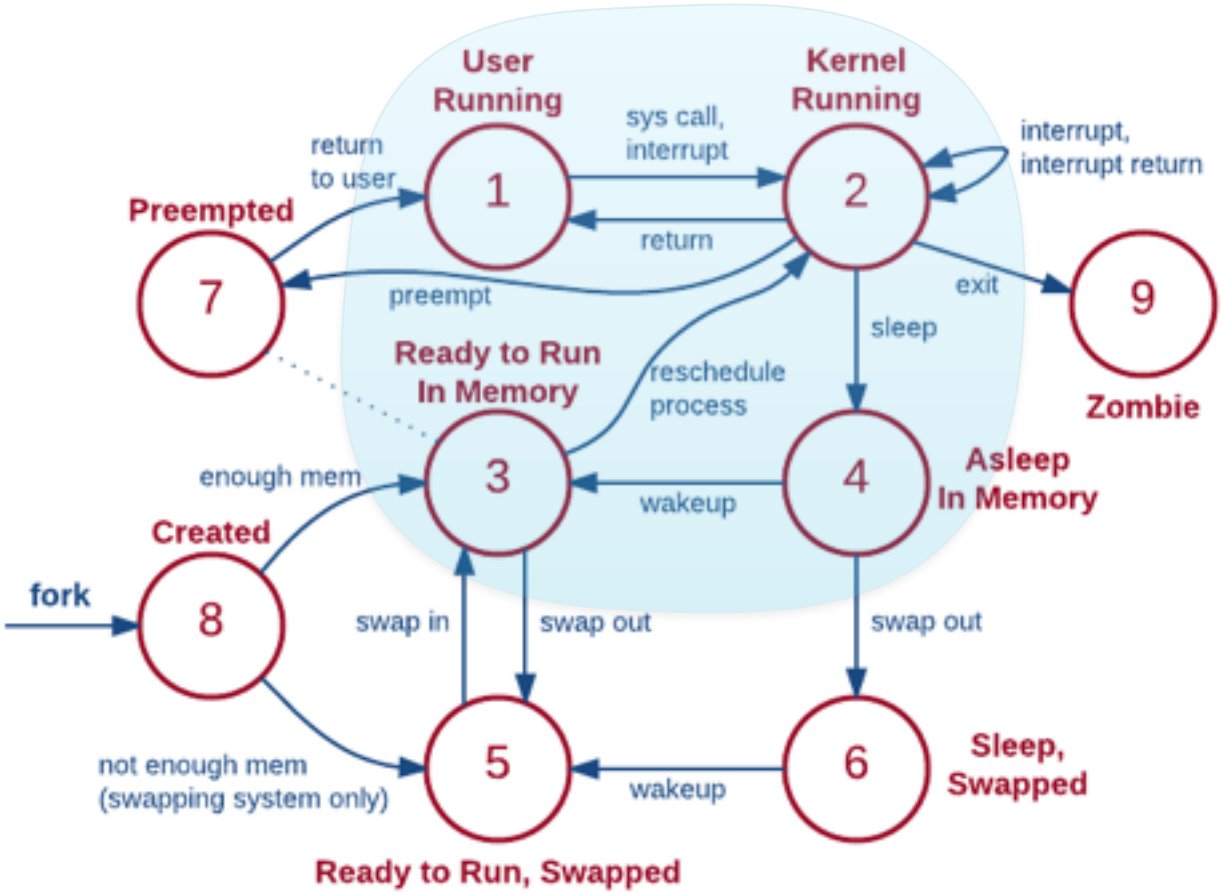


Process State Transition Diagram

Process States (15)

- When the process completes the system call, it may move to the state “user running”
- After the period of time the system clock may interrupt the processor and the process will enter “kernel running” state again
- The kernel may decide to schedule another process to execute, so the first process will enter “preempted” state and the other process executes
- The state “preempted” is the same as the state “ready to run in memory”, but it is important to understand that **a process executing in kernel mode can be preempted only when it is about to return to user mode**

Process States (16)

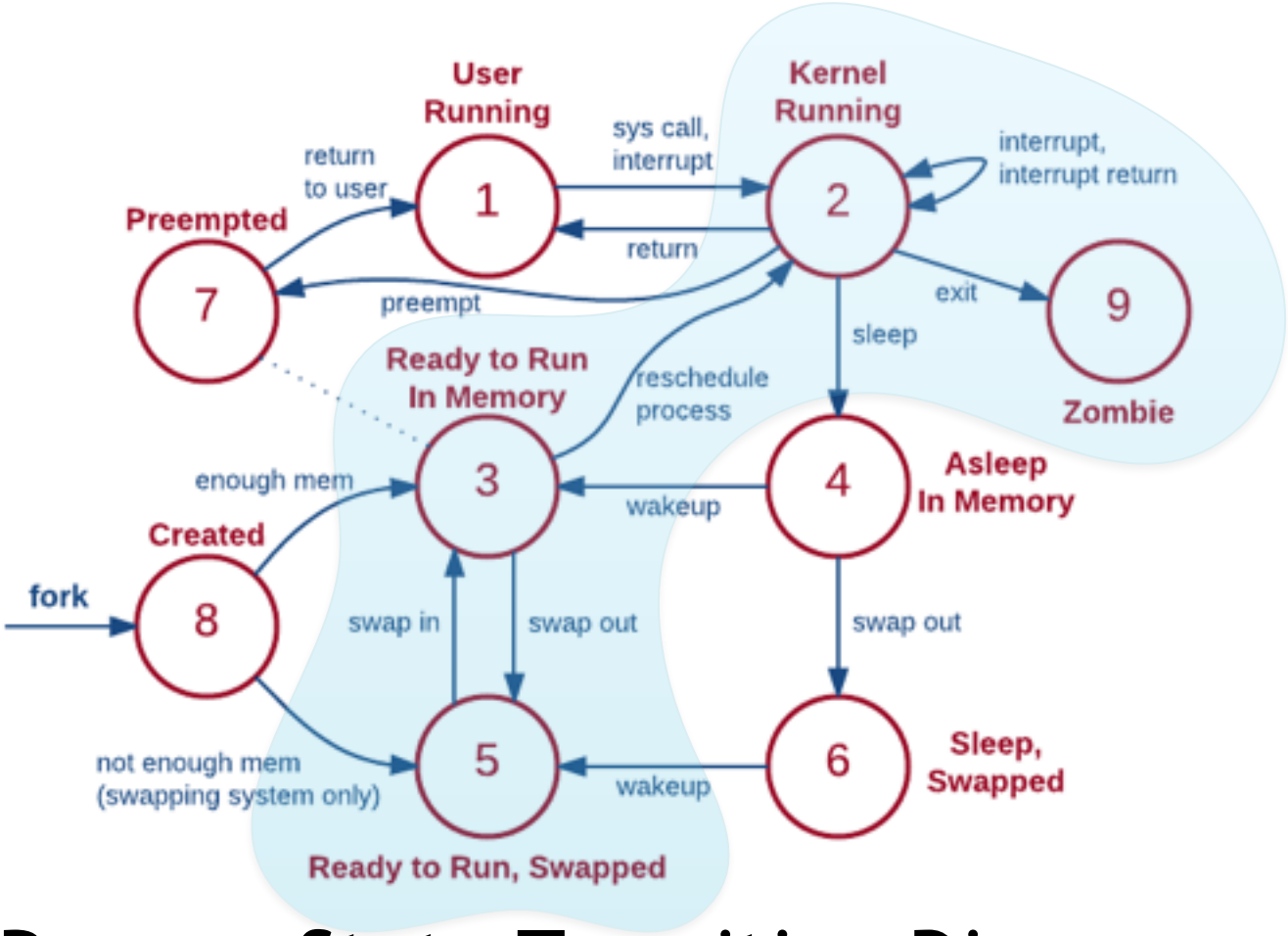


Process State Transition Diagram

Process States (17)

- When a process executes a system call, it leaves the state "user running" and enters the state "kernel running”
- Suppose the system call requires I/O from the disk, and the process must wait for the I/O to complete. It enters the state "asleep in memory," and sleeps until it is notified that the I/O has completed. When it happens, the hardware interrupts the CPU, and the interrupt handler awakens the process, causing it to enter the state "ready to run in memory"

Process States (18)



Process State Transition Diagram

Process States (19)

- Suppose the system is executing many processes that do not fit simultaneously into main memory, and the swapper (process 0) swaps out the process to make room for another process that is in the state "ready to run swapped”
- When evicted from main memory, the process enters the state "ready to run swapped"
- Eventually, the swapper chooses the process as the most suitable to swap into main memory, and the process reenters the state "ready to run in memory"
- The scheduler will eventually choose to run the process, and it enters the state "kernel running" and proceeds
- When a process completes, it invokes the exit system eau, thus entering the states "kernel running" and, finally, the "zombie" state

Implementation of Processes (1)

- To implement the process model, the OS maintains a table (an array of structures), called the **process table**, with one entry per process (sometimes called a **process control block**)
- This entry contains information about the process' state that must be saved when the process is switched from running to ready or blocked state, including:
  - its program counter
  - stack pointer
  - memory allocation
  - the status of its open files
  - its accounting and scheduling information, etc.
- So that it can be restarted later as if it had never been stopped

Implementation of Processes (2)

Process management	Memory management	File management
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Figure 2-4. Some of the fields of a typical process table entry.

Implementation of Processes (3)

- Associated with each I/O class is a location called the **interrupt vector,** which contains the address of the interrupt service procedure
- Suppose that user process 3 is running when a disk interrupt happens:
  - User process 3's program counter, program status word, and sometimes one or more registers are pushed onto the stack by the interrupt hardware
  - The computer then jumps to the address specified in the interrupt vector. That is all the hardware does
  - From here on, it is up to the software, in particular, the interrupt service procedure

Implementation of Processes (4)

- All interrupts start by saving the registers, often in the process table entry for the current process
- Then the information pushed onto the stack by the interrupt is removed and the stack pointer is set to point to a temporary stack used by the process handler
- Actions such as saving the registers and setting the stack pointer are performed by a small assembly-language routine, usually the same one for all interrupts since the work of saving the registers is identical

Implementation of Processes (5)

- When this routine is finished, it calls a C procedure to do the rest of the work for this specific interrupt type
- When it has done its job the scheduler is called to see who to run next
- After that, control is passed back to the assembly-language code to load up the registers and memory map for the now-current process and start it running
- The key idea is that after each interrupt the interrupted process returns to precisely the same state it was in before the interrupt occurred

Implementation of Processes (6)

1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly language procedure starts up new current process.

Figure 2-5. Skeleton of what the lowest level of the operating system does when an interrupt occurs.

Modeling Multiprogramming (1)

- A better model is to look at CPU usage from a probabilistic viewpoint:
  - Suppose that a process spends a fraction *p* of its time waiting for I/O to complete
  - With *n* processes in memory at once, the probability that all *n* processes are waiting for I/O (means CPU is idle) is p^n.
  - The CPU utilization is then given by the formula:

$$\textbf{CPU utilization} = 1 - p^n$$

- The CPU utilization as a function of *n* is called the **degree of multiprogramming** (Fig. 2-6)

Modeling Multiprogramming (2)

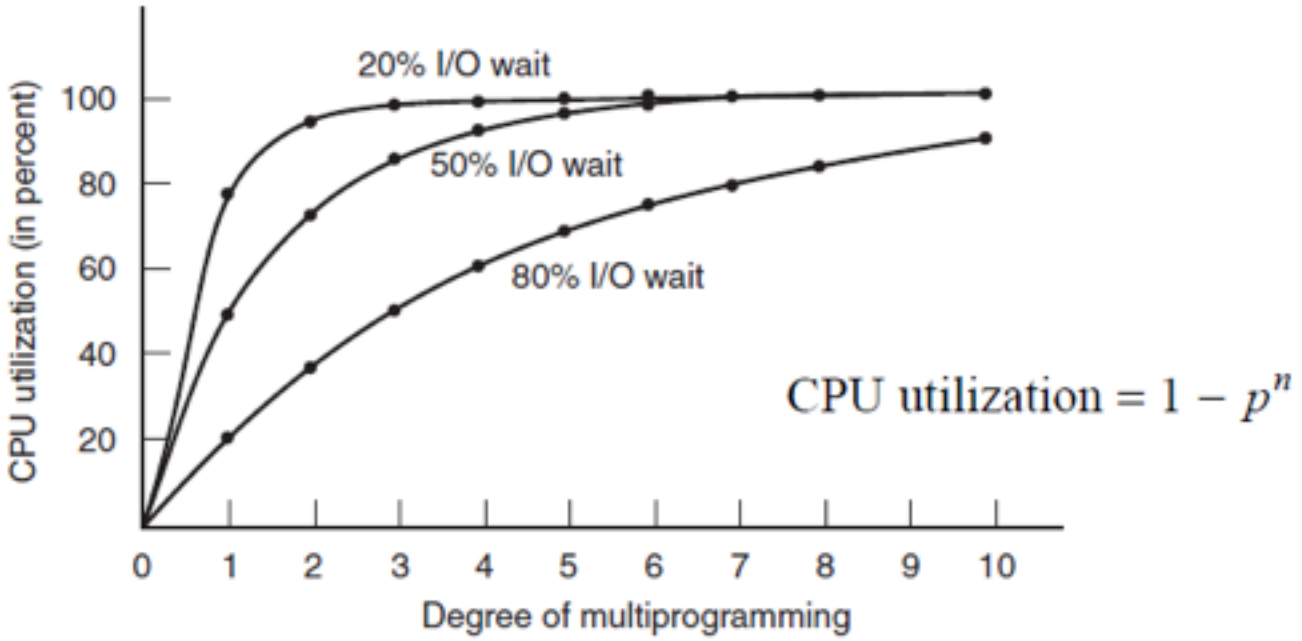


Figure 2-6. CPU utilization as a function of the number of processes in memory.

Modeling Multiprogramming (3)

- If processes spend 80% of their time waiting for I/O, at least 10 processes must be in memory at once to get the CPU waste below 10%
- This probabilistic model is only an approximation – it implicitly assumes that all *n* processes are independent
- We cannot have three processes running at once, so a process becoming ready while the CPU is busy will have to wait. Thus the processes are not independent
- Nevertheless this model can be used to make specific, although approximate, predictions about CPU performance

Modeling Multiprogramming (4)

- Suppose that a computer has 8 GB of memory, with the OS and its tables taking up 2 GB and each user program also taking up 2 GB. Only three user programs can be in memory at once
- With an 80% average I/O wait, we have a CPU utilization (ignoring operating system overhead) of $1 - 0.8^3$ or about 49%
- Adding 8 GB of memory allows the system to go from three-way multiprogramming to seven-way multiprogramming, thus raising the CPU utilization to 79%
- Adding yet another 8 GB would increase CPU utilization only from 79% to 91%, thus raising the throughput by only another 12%
- The first addition was a good investment but that the second was not

End

Week 04 – Lecture 1

References

- Tanenbaum & Bo, Modern Operating Systems: 4th edition, 2013 Prentice-Hall, Inc.