# Processes and Threads

Week 04 – Lecture

Fundamentals

# Team

- Instructor
  - Giancarlo Succi
- Teaching Assistants
  - Nikita Lozhnikov (also Tutorial Instructor)
  - Manuel Rodriguez
  - Shokhista Ergasheva

# Sources

- These slides have been adapted from the original slides of the adopted book:
  - Tanenbaum & Bos, 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 (1/2)

- 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 (2/2)

- 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 the processes
- Such an illusion of parallelism is called **pseudo-parallelism**

# The Process Model (1/6)

- 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 (2/6)

- 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 (3/6)

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

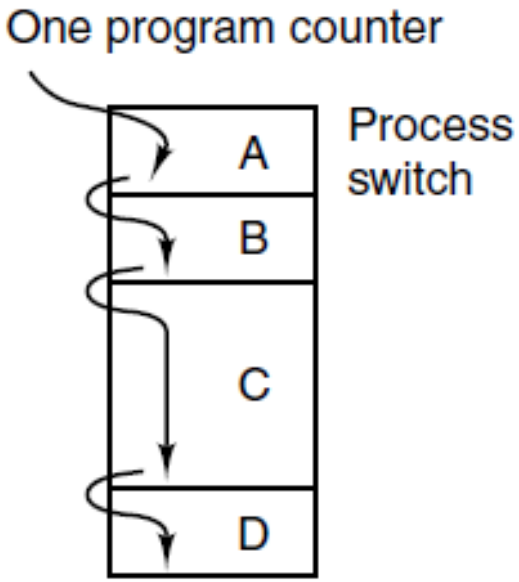



Figure 2-1. (a) Multiprogramming of four programs.

# The Process Model (4/6)

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

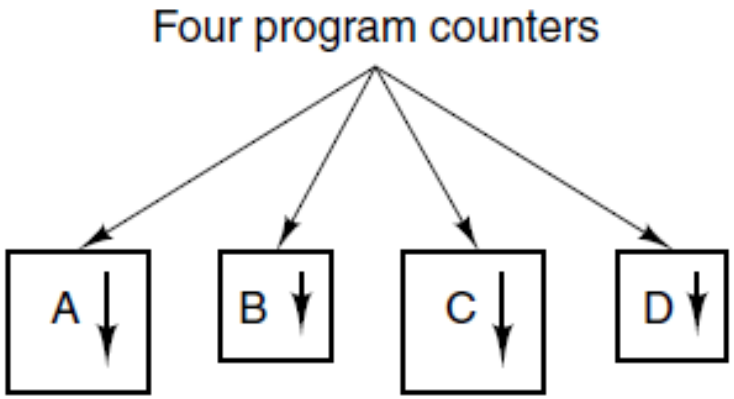



Figure 2-1. (b) Conceptual model of four independent, sequential processes.

# The Process Model (5/6)

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

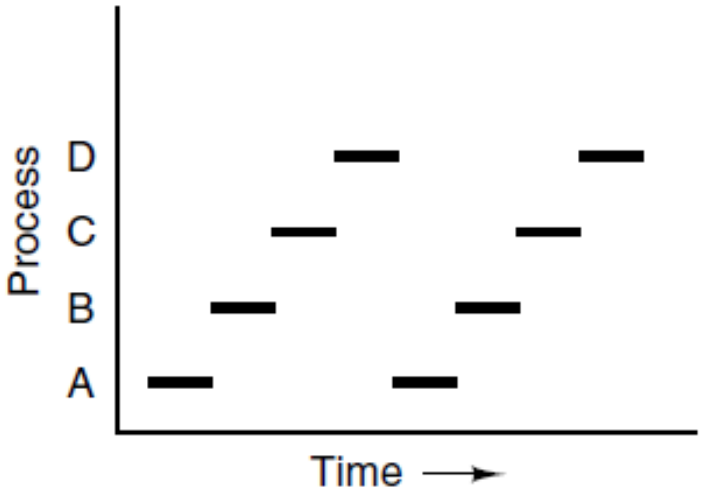



Figure 2-1. (c) Only one program is active at once.

# The Process Model (6/6)

- 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 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 (1/2)

- 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 (2/2)

- 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 Termination

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

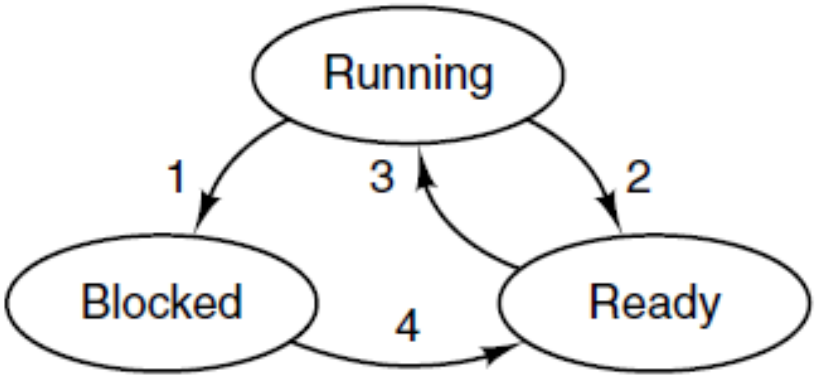
# Process Hierarchies

- 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 States (1/18)

- 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 (2/18)




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 (3/18)

- Transition 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
- Transition 2 (Running → Ready):
  - occurs when the scheduler decides that it is time to let another process have some CPU time

# Process States (4a/18)

- Transition 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
- Transition 4 (Blocked → Ready):
  - occurs when the external event for which a process was waiting (such as the arrival of some input) happens

# Process States (4b/18)

- Comment
  - Transition 4 (Blocked → Ready → Running):
    - *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 (5/18)

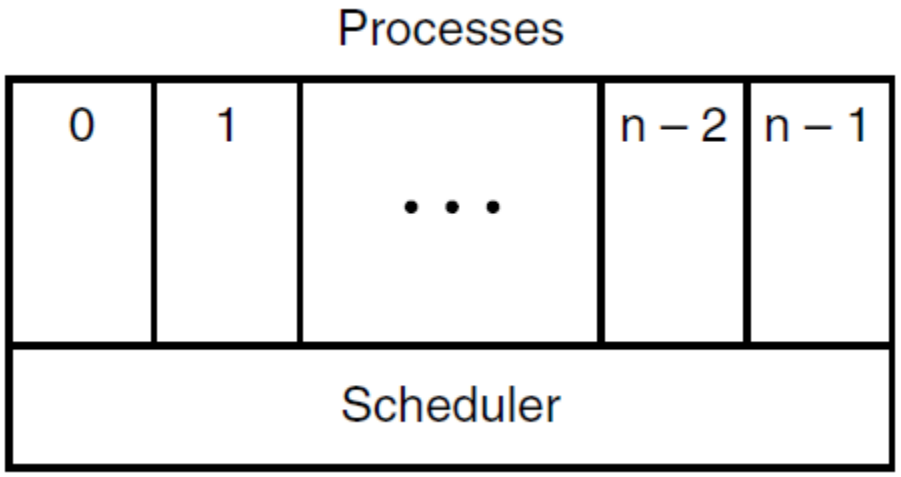
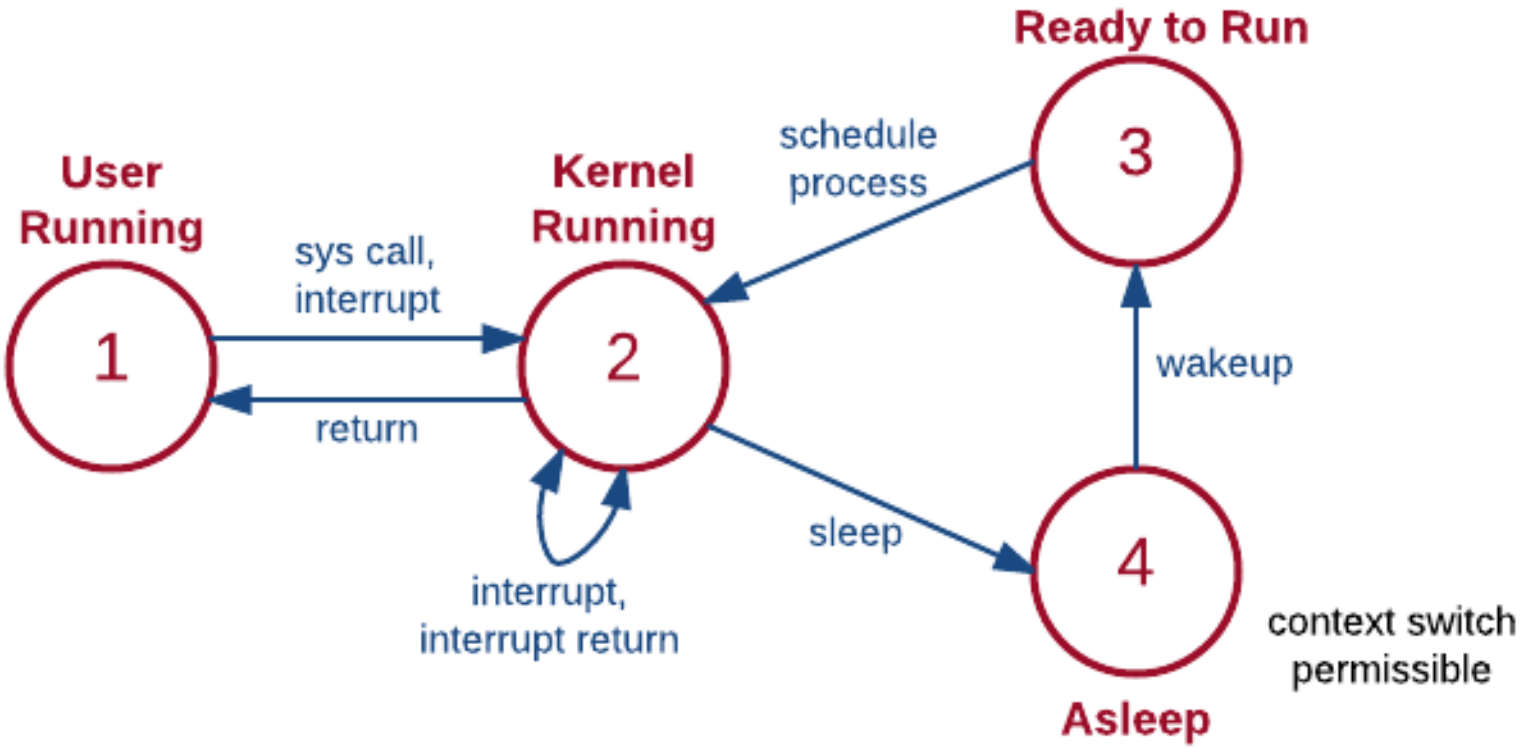



Figure 2-3. The lowest layer of a process-structured operating system handles interrupts and scheduling. Above that layer are sequential processes.

# Process States (6/18)

- 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 (7/18)




Process States and Transitions

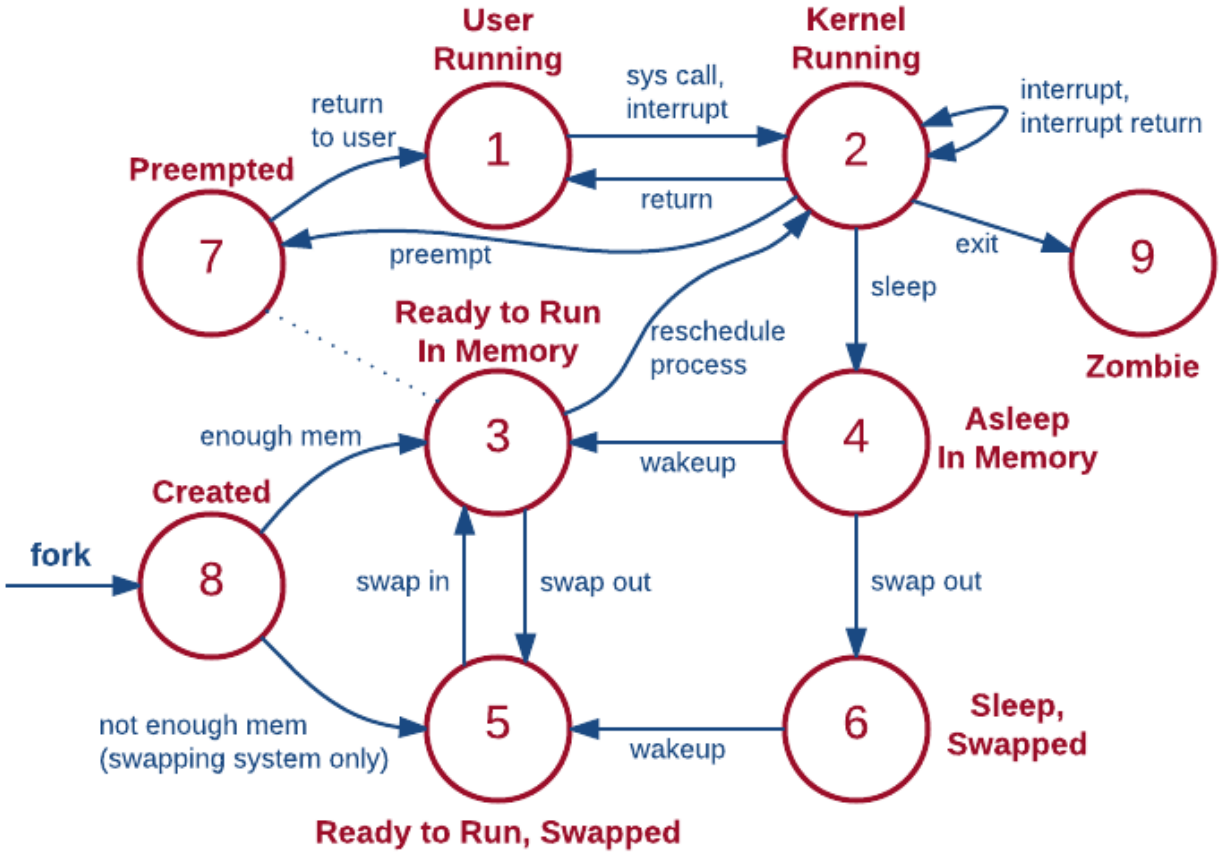
# Process States (8/18)

- 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, but the swapper (process 0) must swap the process into main memory before the kernel can schedule it to execute

# Process States (9/18)

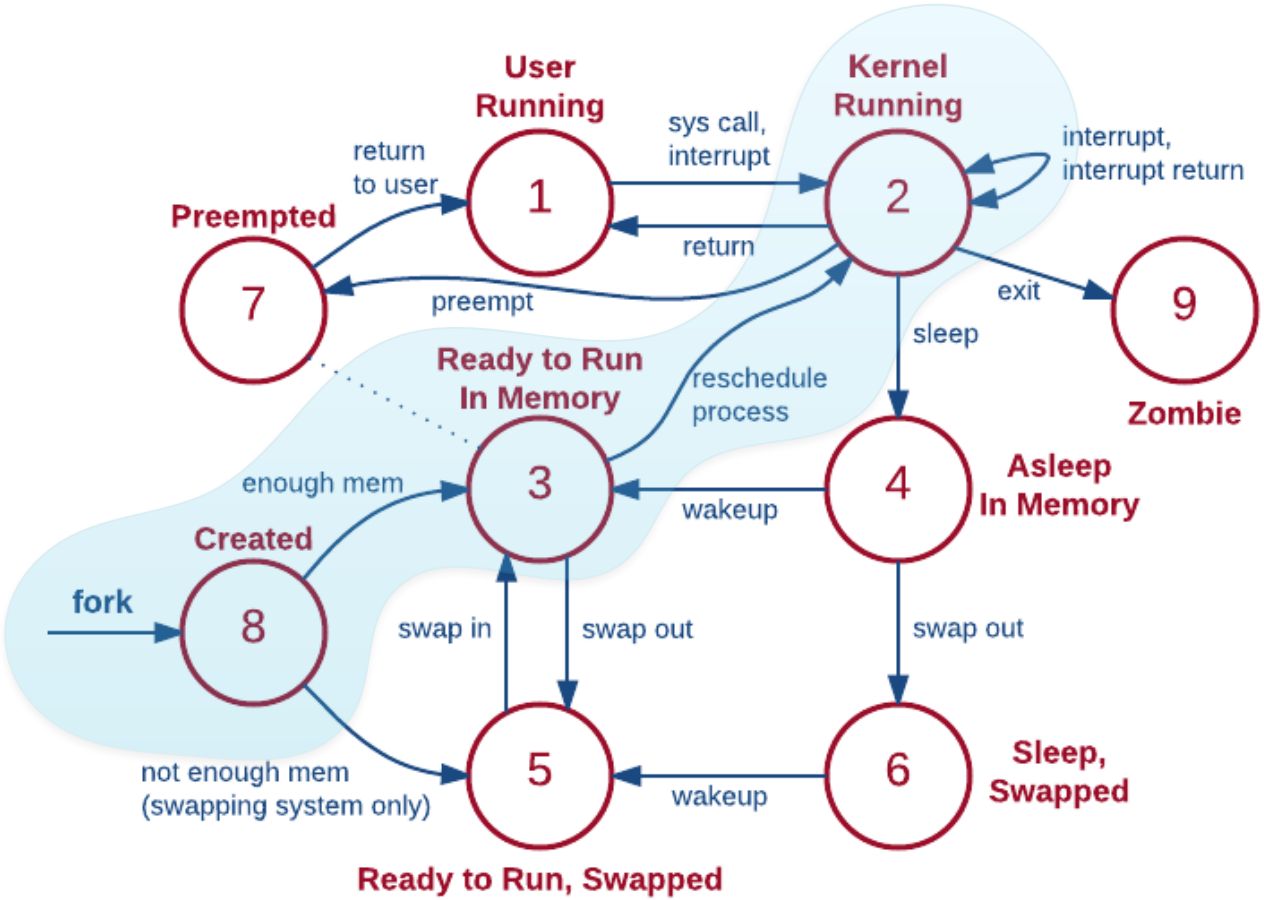
6. The process is awaiting an event and has been swapped to secondary storage (a blocked state)
7. The process is returning from kernel to user mode, but the kernel preempts it and does a context switch to schedule another process
8. The process is newly created and is in a transition state; it is not 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 (10/18)




Process State Transition Diagram

# Process States (11/18)

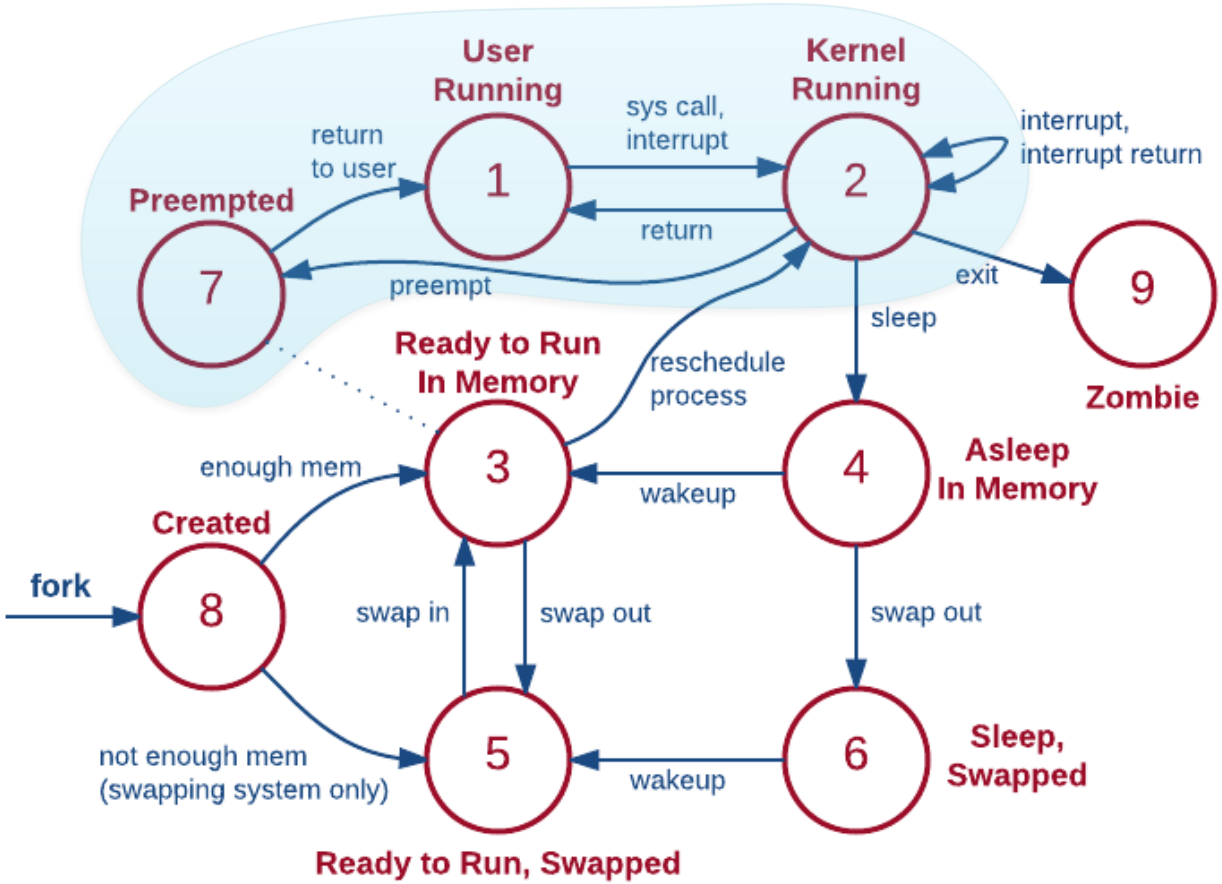



Process State Transition Diagram

# Process States (12/18)

- 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 (13/18)

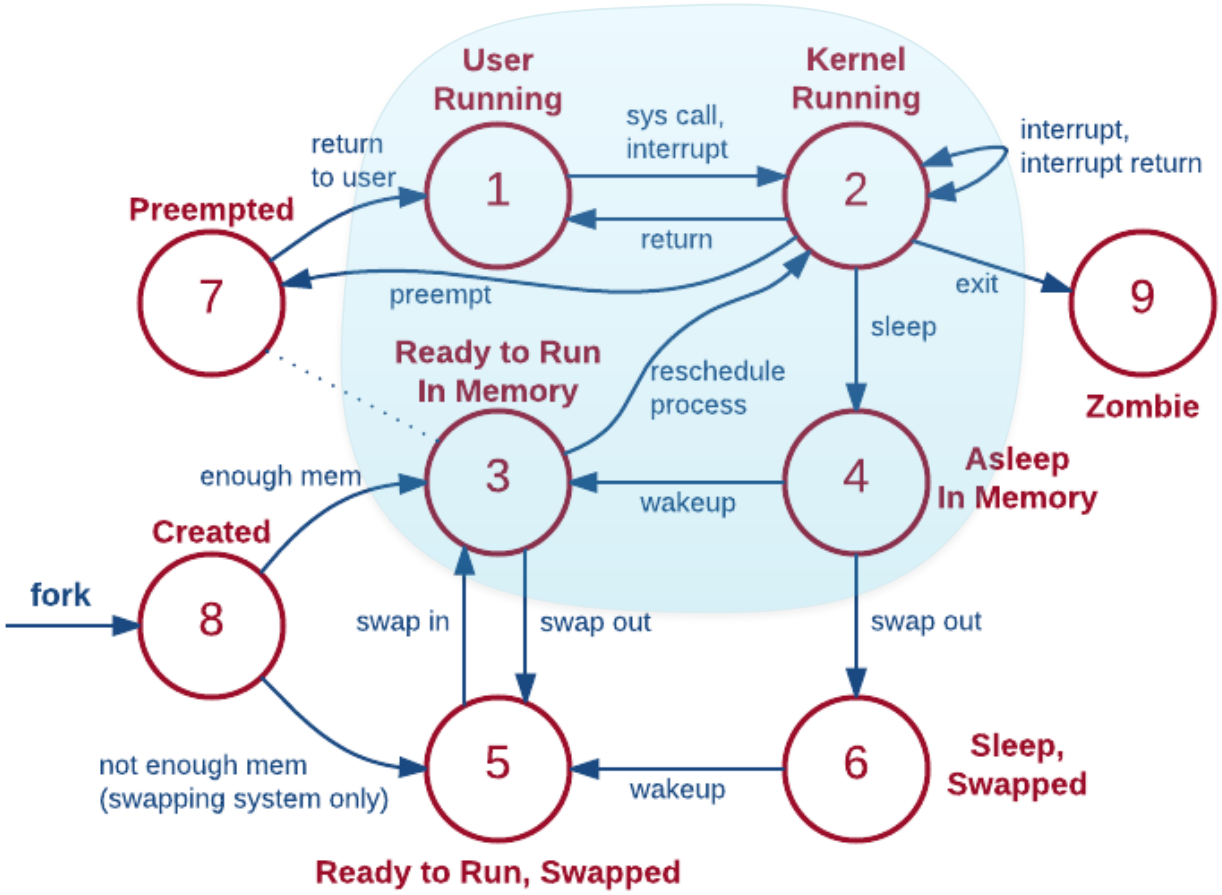



Process State Transition Diagram

# Process States (14/18)

- 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 (15/18)

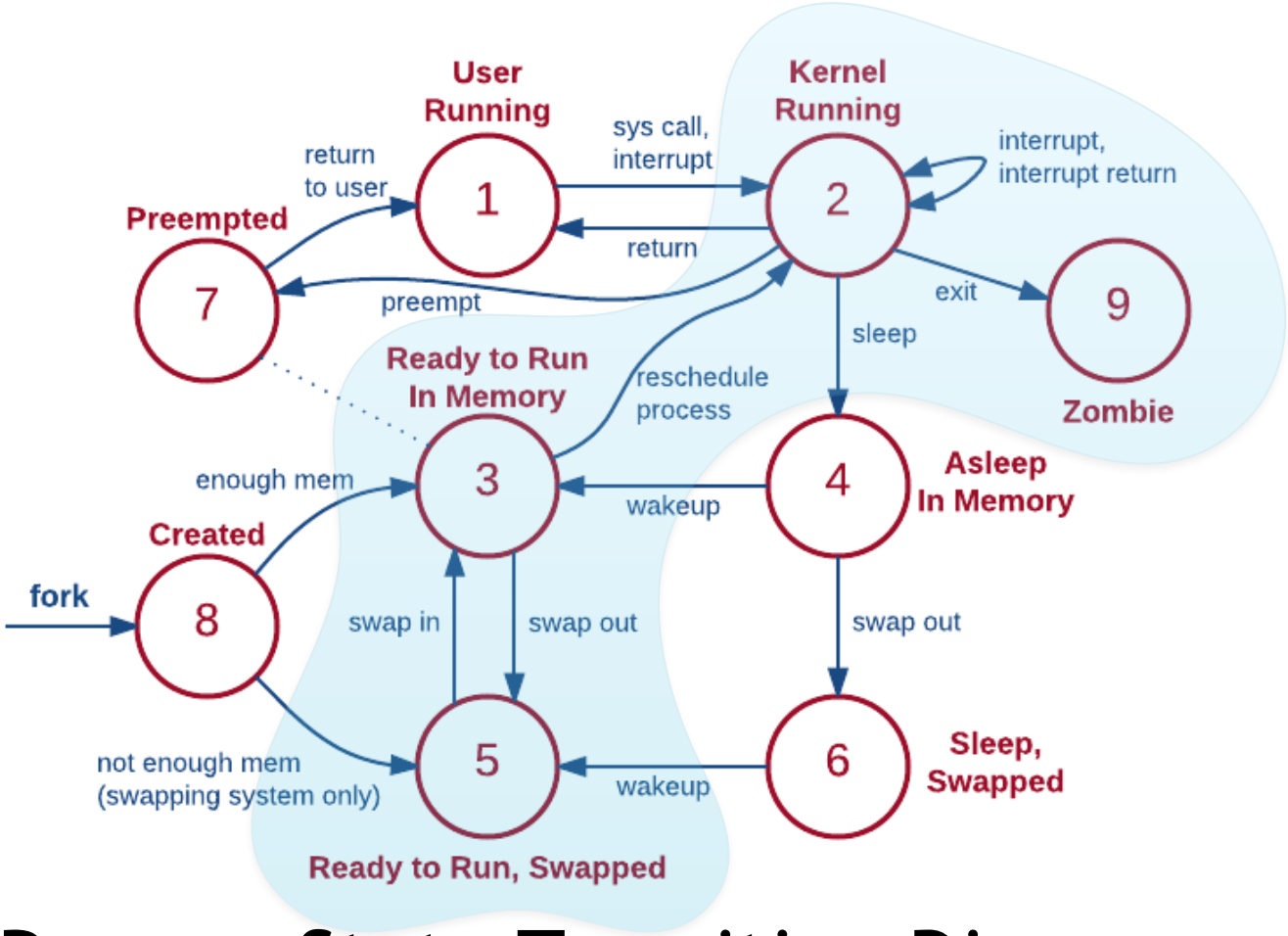



Process State Transition Diagram

# Process States (16/18)

- 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 (17/18)




Process State Transition Diagram

# Process States (18/18)

- Many processes might 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”
- The process enters the state "ready to run swapped"
- Eventually, 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 call, thus entering the states "kernel running" and, finally, the "zombie" state

# Implementation of Processes (1/3)

- 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:
  - its program counter
  - stack pointer
  - memory allocation
  - the status of its open files
  - its accounting and scheduling information, etc.

# Implementation of Processes (2/3)

- 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 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 (3/3)

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 (1/2)

- 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:

**CPU utilization = 1 – $p^n$**

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

# Modeling Multiprogramming (2/2)

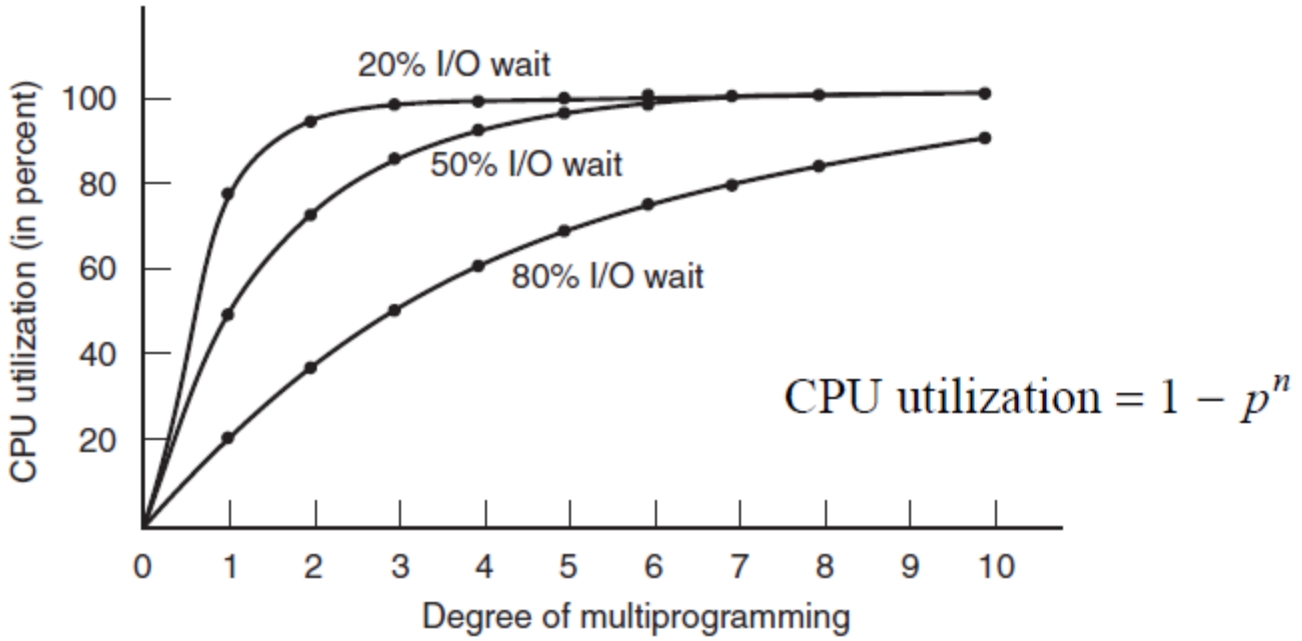



Figure 2-6. CPU utilization as a function of the number of processes in memory.

# Threads

- In traditional OSs each process has an address space and a **single thread of control**
- In many situations, it is desirable to have **multiple threads of control** in the same address space running in quasi-parallel, as though they were separate processes (except for the shared address space)

# Thread Usage (1/6)

- There are several reasons for having these miniprocesses, called threads:
  - Decomposing an application into multiple sequential threads that run in quasi-parallel, the programming model becomes simpler
  - easier (i.e., faster) to create and destroy than processes
  - having threads allows I/O and computing to overlap
  - threads with multiple CPUs provide real parallelism

# Thread Usage (2/6)

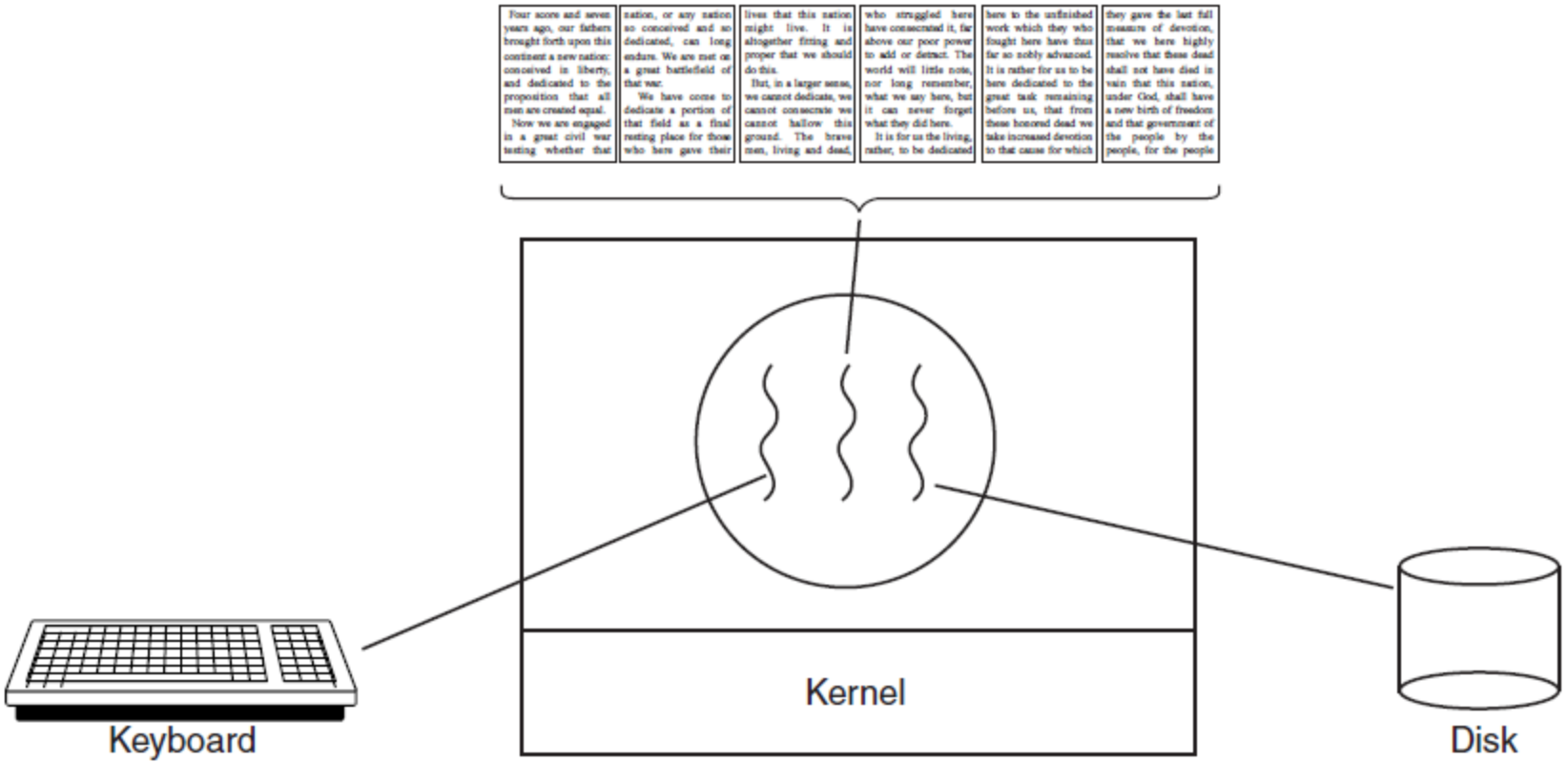



Figure 2-7. A word processor with three threads.

# Thread Usage (3/6)

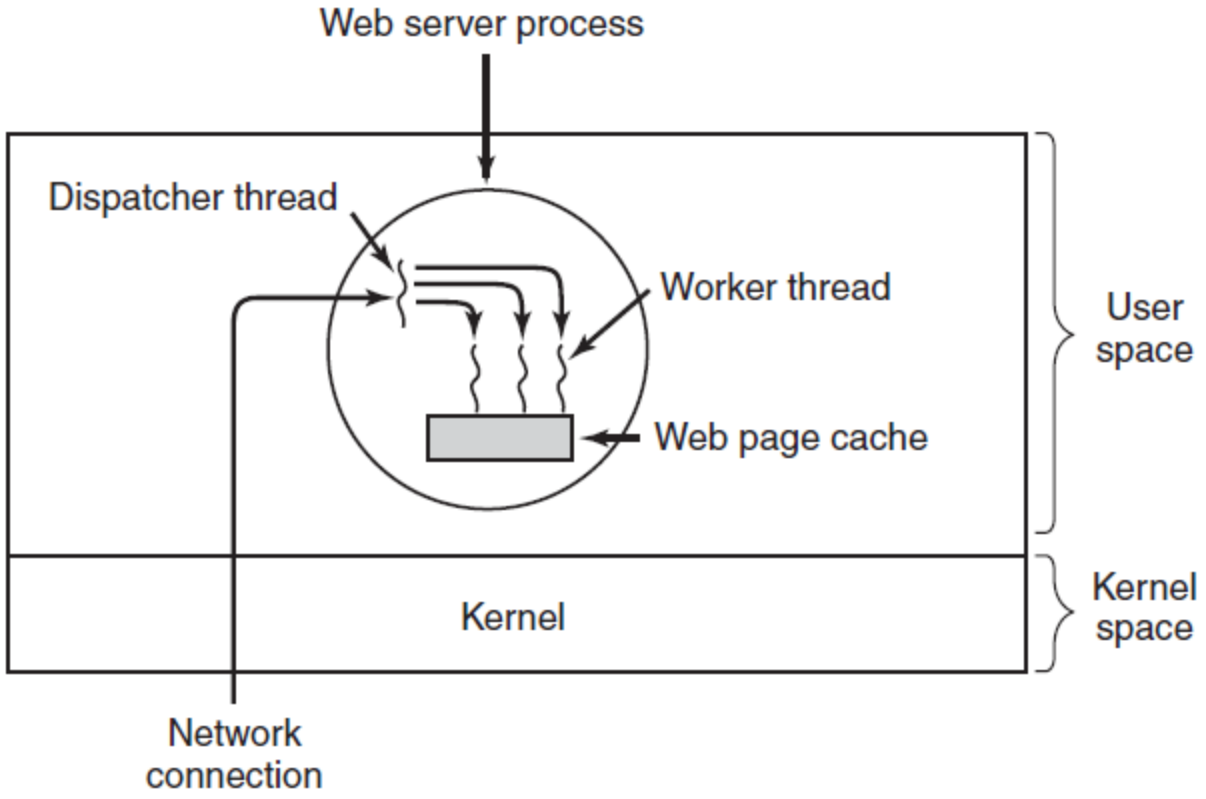



Figure 2-8. A multithreaded Web server.

# Thread Usage (4/6)

- This model allows the server to be written as a collection of sequential threads (Fig. 2-9)
- The dispatcher's program consists of an infinite loop for getting a work request and handing it off to a worker
- Each worker's code consists of an infinite loop consisting of accepting a request from the dispatcher and checking the cache to see if the page is present
  - If so, it is returned to the client, and the worker blocks waiting for a new request
  - If not, it gets the page from the disk, returns it to the client, and blocks waiting for a new request

# Thread Usage (5/6)

```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

(a)

```
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

(b)

Figure 2-9. A rough outline of the code for Fig. 2-8. (a) Dispatcher thread. (b) Worker thread.

# Thread Usage (6/6)

| Model | Characteristics |
|---|---|
| Threads | Parallelism, blocking system calls |
| Single-threaded process | No parallelism, blocking system calls |
| Finite-state machine | Parallelism, nonblocking system calls, interrupts |

Figure 2-10. Three ways to construct a server.

# The Classical Thread Model (1/10)

- The process model is based on two independent concepts: resource grouping and execution
- Sometimes it is useful to separate them; this is where threads come in
- First we will look at the classical thread model; after that we will examine the Linux thread model, which blurs the line between processes and threads

# The Classical Thread Model (2/10)

- A process has an address space containing program text and data, as well as other resources
- These resources may include:
  - open files
  - child processes
  - pending alarms
  - signal handlers
  - accounting information, and more
- By putting them together in the form of a process, they can be managed more easily

# The Classical Thread Model (3/10)

- The other concept a process has is a **thread of execution** or just **thread**
- The thread has:
  - a program counter that keeps track of which instruction to execute next
  - registers, which hold its current working variables
  - a stack, which contains the execution history, with one frame for each procedure called but not yet returned from

# The Classical Thread Model (4/10)

- Because threads have some of the properties of processes, they are sometimes called **lightweight processes**
- The term **multithreading** is also used to describe the situation of allowing multiple threads in the same process
- Some CPUs have direct hardware support for multithreading and allow thread switches to happen in nanoseconds

# The Classical Thread Model (5/10)

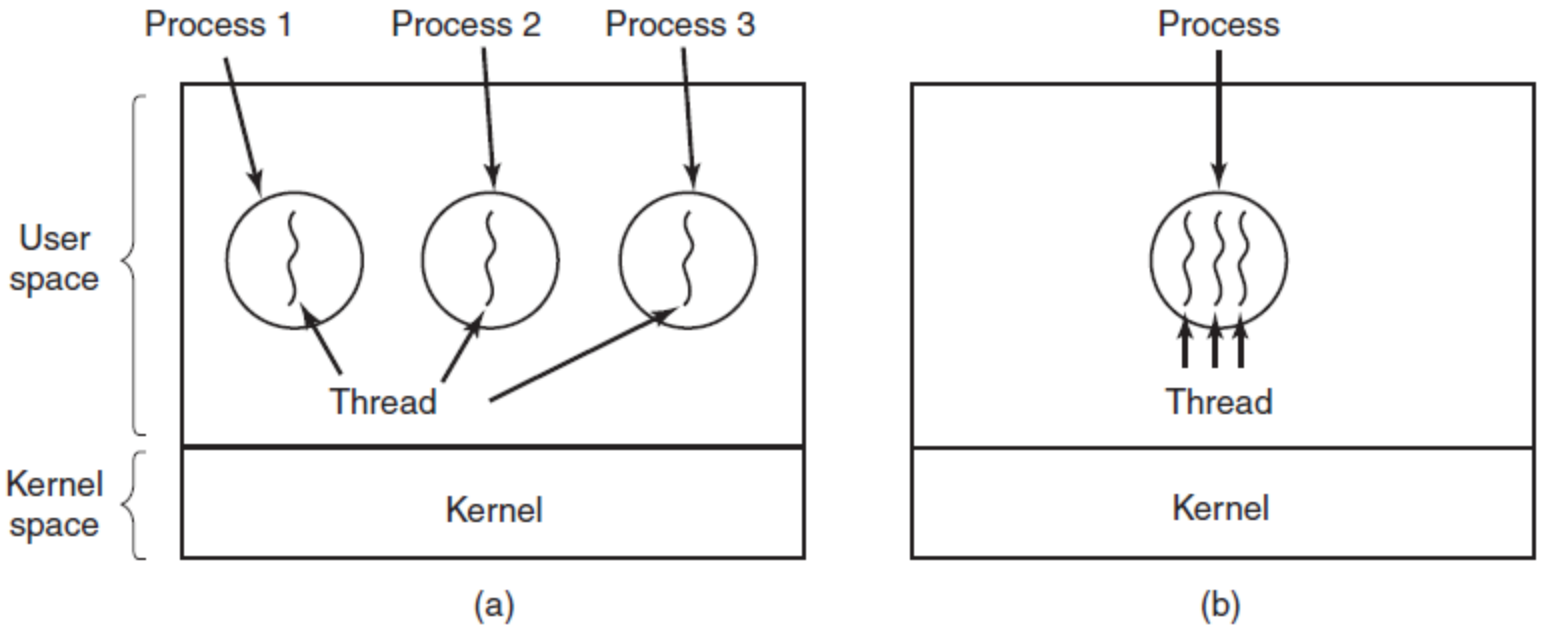



Figure 2-11. (a) Three processes each with one thread. (b) One process with three threads.

# The Classical Thread Model (6/10)

| Per process items | Per thread items |
|---|---|
| Address space | Program counter |
| Global variables | Registers |
| Open files | Stack |
| Child processes | State |
| Pending alarms | |
| Signals and signal handlers | |
| Accounting information | |

Figure 2-12. The first column lists some items shared by all threads in a process. The second one lists some items private to each thread.

# The Classical Thread Model (7/10)

- Like a traditional single-threaded process, a thread can be in any one of several states:
  - A **running** thread currently has the CPU and is active
  - A **blocked** thread is waiting for some event to unblock it
  - A **ready** thread is scheduled to run and will as soon as its turn comes up
  - Thread can be also **terminated**
- The transitions between thread states are the same as those between process states

# The Classical Thread Model (8/10)

- Each thread has its own stack (Fig. 2-13) that contains one frame for each procedure called but not yet returned from
- A frame contains the procedure's local variables and the return address to use when the procedure call has finished
- For example, if procedure *X* calls procedure *Y* and *Y* calls procedure *Z*, then while *Z* is executing, the frames for *X*, *Y*, and *Z* will all be on the stack

# The Classical Thread Model (9/10)

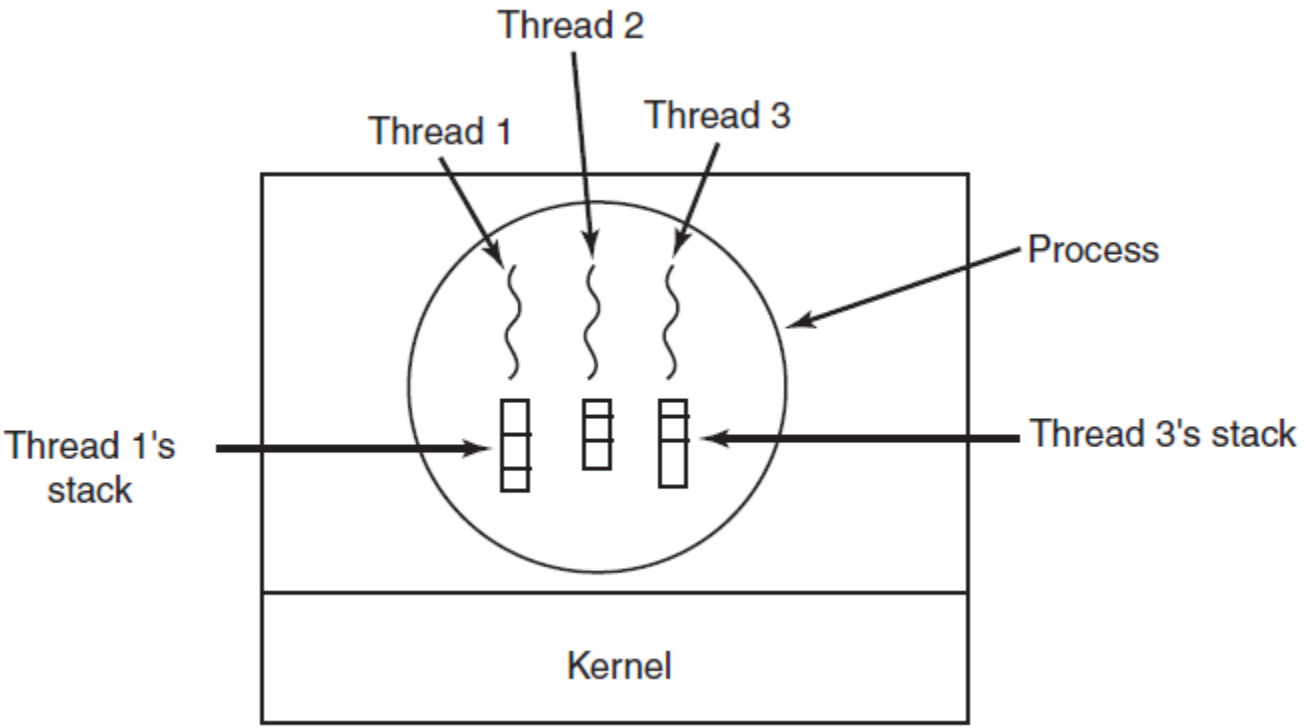



Figure 2-13. Each thread has its own stack.

# The Classical Thread Model (10/10)

- Using threads brings a number of complications into the programming model:
  - If the parent process has multiple threads, should the child also have them?
  - What happens if a thread in the parent was blocked on a *read* call from the keyboard?
  - When a line is typed, do both threads get a copy of it?
  - What happens if one thread closes a file while another one is still reading from it?

# Making Single-Threaded Code Multithreaded (1/5)

- Many existing programs were written for single-threaded processes. Converting these to multithreading is tricky. Some of the pitfalls are:
  - having to deal with the variables that a global to the process (Fig. 2-11)
  - many library procedures are not reentrant
  - dealing with non-thread-specific signals (or even with thread-specific signals in user space)
  - stack management

# Making Single-Threaded Code Multithreaded (2/5)

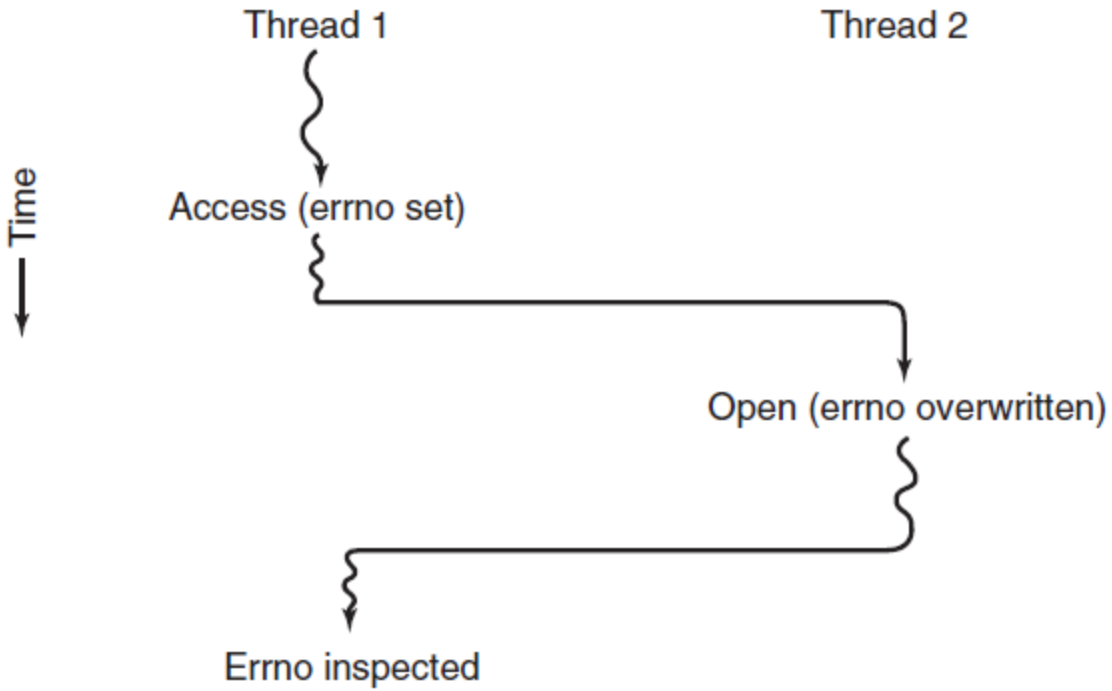



Figure 2-19. Conflicts between threads over the use of a global variable.

# Making Single-Threaded Code Multithreaded (3/5)

- One of the solutions to the first problem is to prohibit global variables
- Another one is to introduce private global variables specific for each thread (Fig. 2-20)
- The third one is to create special library procedures for creating, setting and reading threadwide global variables, such as:
  - create_global(“bufptr");
  - set_global(“bufptr", &buf);
  - bufptr = read_global(“bufptr");

# Making Single-Threaded Code Multithreaded (4/5)

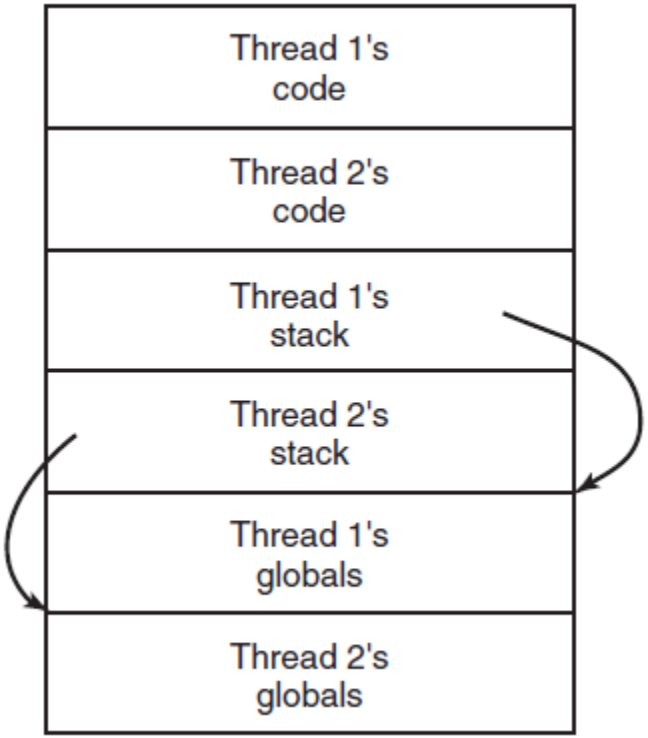



Figure 2-20. Threads can have private global variables.

# Making Single-Threaded Code Multithreaded (5/5)

- The other problems are harder to deal with
- For example, to solve the second problem, one would need to rewrite the whole library or use a jacket (wrapper) that sets a bit to mark the library as in use and will not allow to call a procedure while this bit is set. Such an approach greatly eliminates potential parallelism
- All of these problems are more or less manageable, but they do show that just introducing threads into an existing system without a fairly substantial system redesign will not work at all

# End

## Week 04 – Lecture

# References


- Tanenbaum & Bos, Modern Operating Systems: 4th edition, 2013 Prentice-Hall, Inc.
- Maurice J. Bach. The Design of the UNIX Operating System, 1986 Prentice Hall