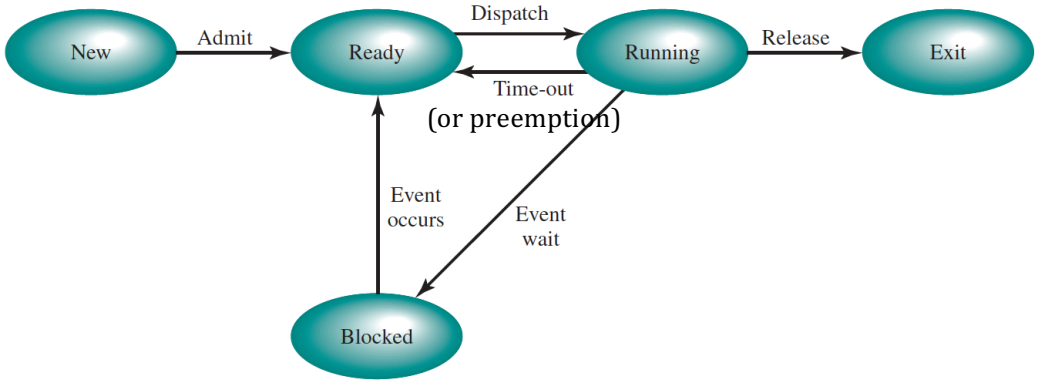
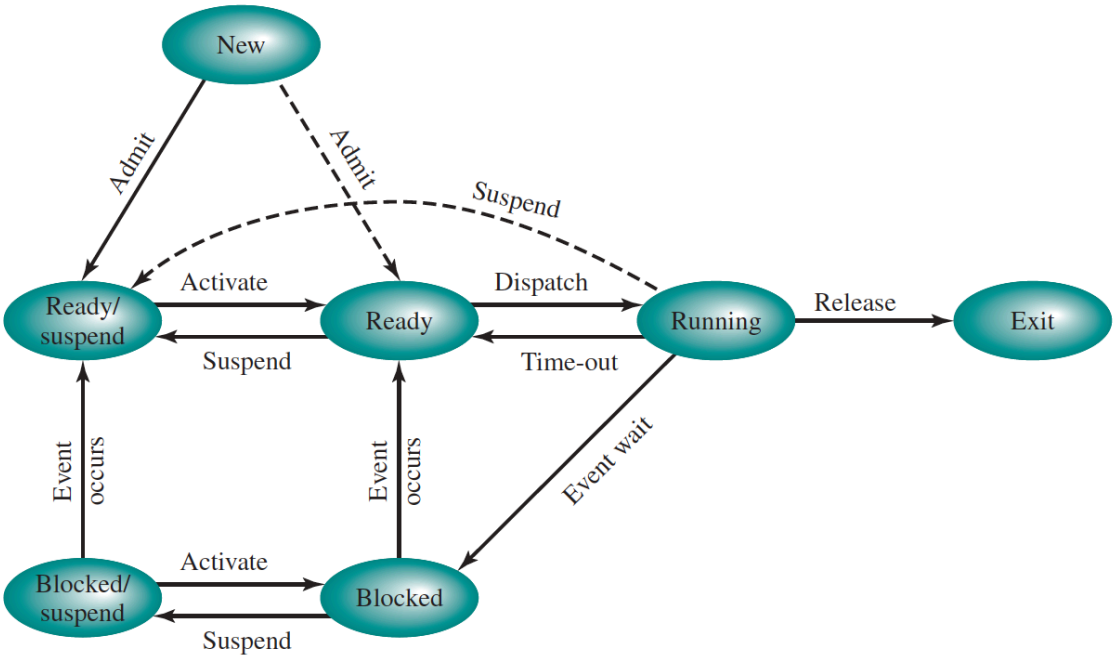
Chapter 3 (3.1 – 3.6)

- **Process**
  - A program in execution
  - An entity that can be assigned to and executed on a processor
  - A unit of activity with thread of execution, a state, and an associated set of resources
  - Consists of:
    - Executable program
    - Associated data used by program
    - Execution context (info needed by OS to manage it)
  - Trace – the sequence of instructions that execute for a process
- **Process control block**:
  - Identifier
  - State information
    - User-visible registers
    - Control/status registers (PC, condition codes, PSW)
    - Stack pointers
  - Control information
    - Scheduling information (state, priority)
    - Data structuring
    - IPC information (messages)
    - Process privileges
    - Memory mgmt., resource ownership/utilization
- **Process image** = PCB + user data + user program + stack
  - Stored in process tables
- **5-state model**:
  - Advantage over two-state:
    - No need to go through entire queue to check if process can be run (blocked or not)
  - Should have one ready & blocked queue for each priority
  - Should have multiple blocked queues, one for each resource – so that OS does not have to scan through all blocked processes when an event occurs



  - Reasons for process creation:
    - New batch job, user log-on, created by OS, or spawned by another process

    - Reasons for process termination:
        - Time limit exceeded, parent termination, or some sort of error
- **Process suspension (7-state model)**:
    - When all processes are blocked, move some to secondary memory
        - Either swap out suspended process or create new process
    - Characteristics of a suspended process:
        - Process is not ready
        - Blocking condition is independent of suspend condition
        - Process was suspended by an agent
        - Process cannot be removed from this state until agent orders it to
    - Reasons for process suspension:
        - Swap to free main memory, user request, time sharing, or parent request

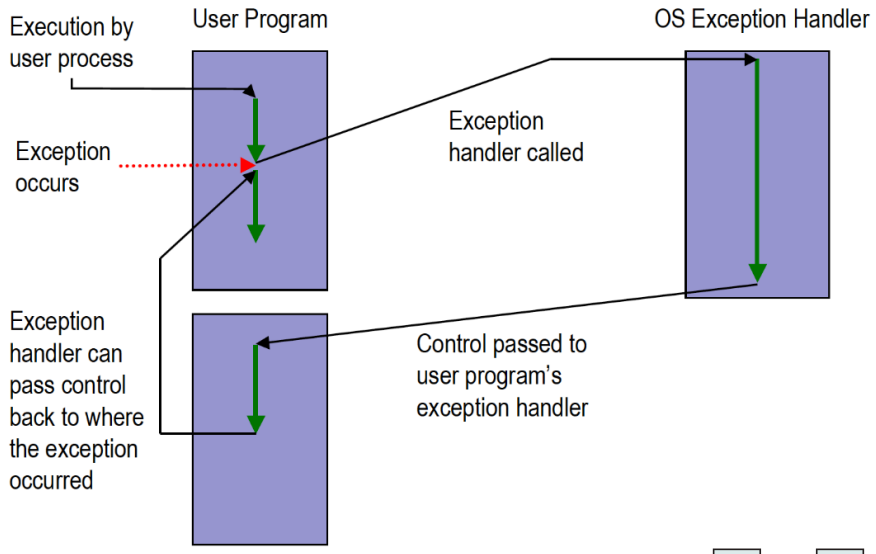


- Mode of execution – <u>user</u> vs. <u>kernel mode</u>
    - A bit in the PSW determines execution mode
- **Steps in process creation**
    - Assign unique id
    - Allocate memory space
    - Initialize PCB
    - Set linkages (e.g. put into ready queue)
    - Create other data structures (e.g. accounting file)
- **Process switching**
    - Possible situations to process switch:

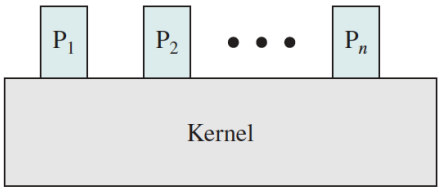
Mechanism	Cause	Use
Interrupt	External to the execution of the current instruction	Reaction to an asynchronous external event
Trap	Associated with the execution of the current instruction	Handling of an error or an exception condition

- Mode switching – used to handle interrupts
  - Set PC to interrupt handler program
  - Change mode bit in PSW to kernel mode
- Process state switching – more overhead than mode switching, done when the current running process needs to be moved to another state
  - Save context of processor (PC, other registers)
  - Update PCB of running process (e.g. change state info)
  - Move PCB to appropriate queue
  - Select another process for execution (i.e. scheduler)
  - Update PCB of selected process (e.g. change state info)
  - Update memory mgmt. data structures
  - Restore context of processor (reload registers)

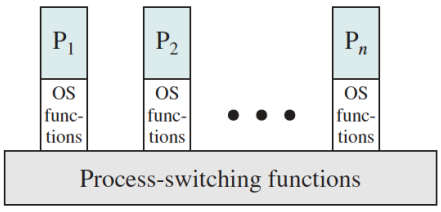
- Exception handling



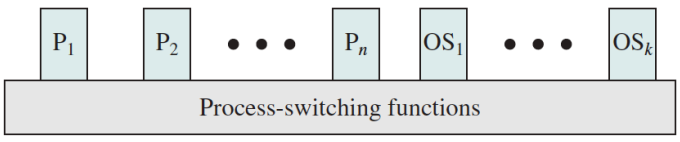
- **Non-process kernel**
  - Kernel has own region of memory; always operates in privileged mode
  - Context switch from process → kernel on interrupts
- **OS functions within user processes**
  - Collection of kernel routines resides in every process
  - Only mode switch is required for interrupts
- **OS functions as separate process**
  - Modular; useful in multiprocessor environment



(a) Separate kernel

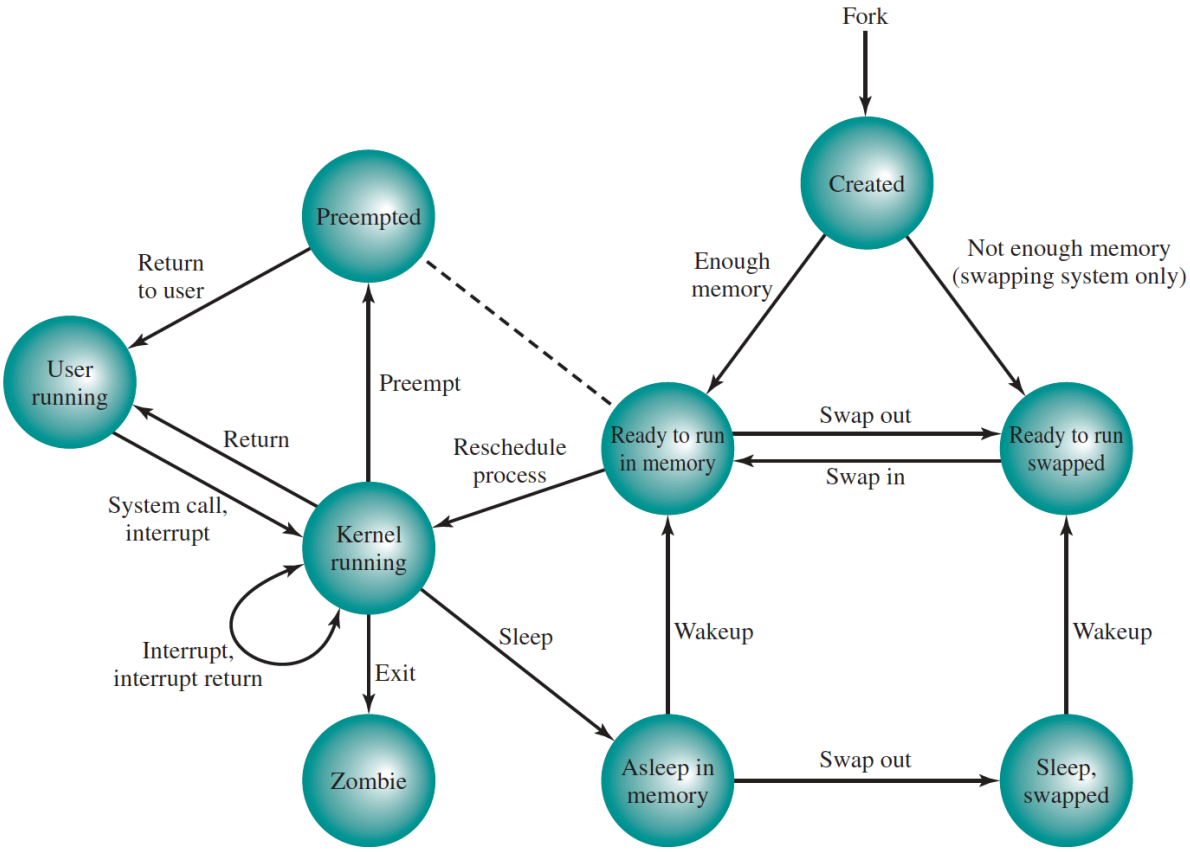


(b) OS functions execute within user processes



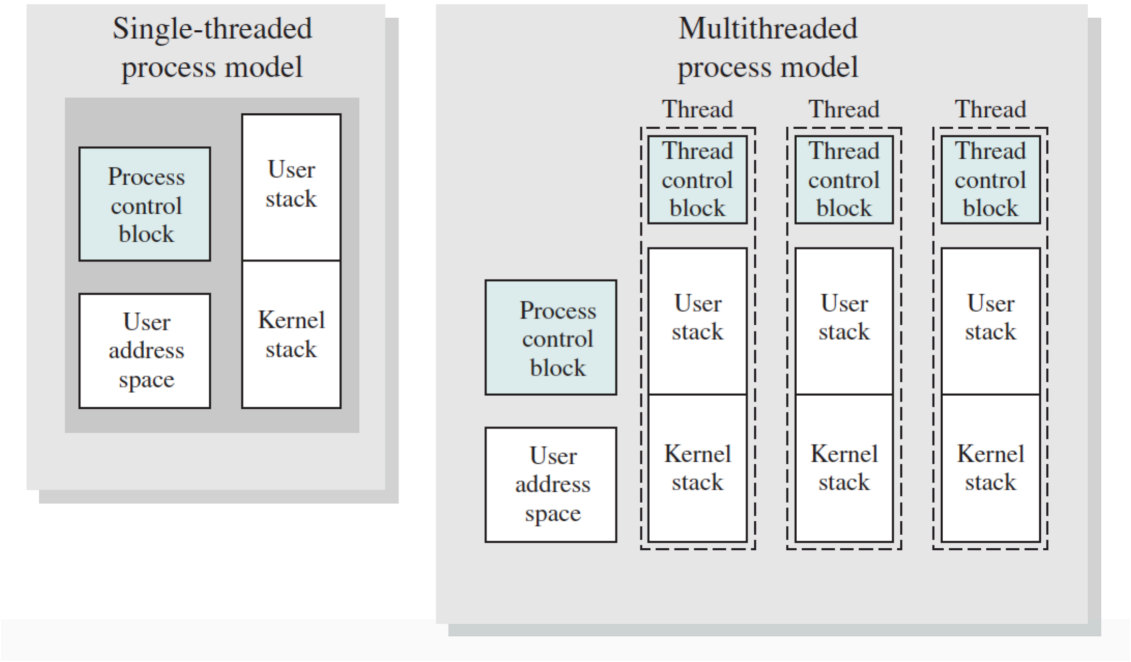
(c) OS functions execute as separate processes

- **UNIX SVR4 process management**
  - Structured mostly as <u>OS functions within user processes</u>
  - System process → kernel mode
  - User processes → user mode; → kernel mode when making system calls or on interrupt
  - Preemption occurs only on kernel → user mode; cannot be preempted in kernel mode



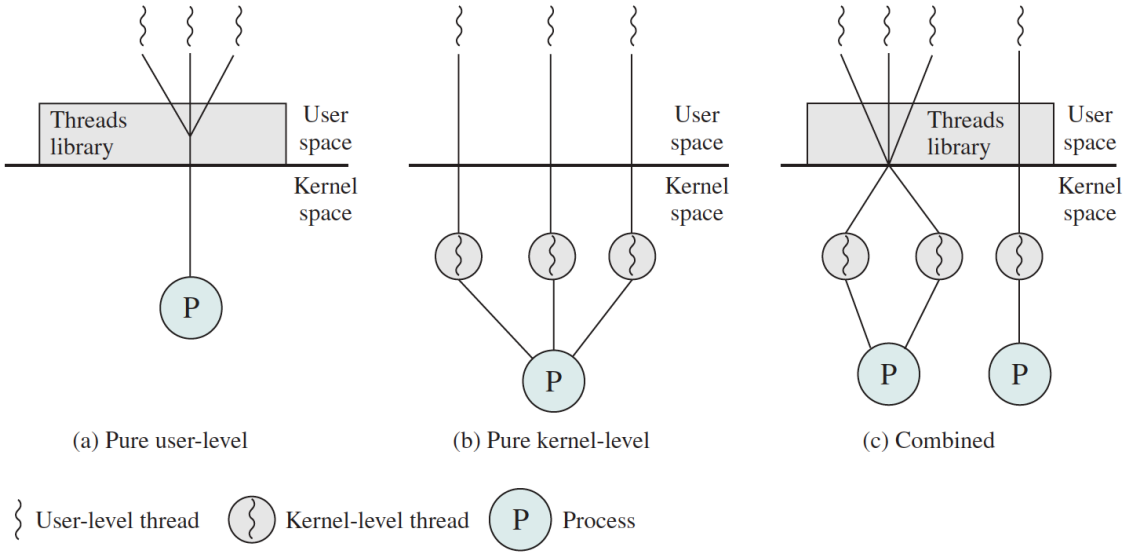
Chapter 4 (4.1 – 4.6)

- **Threads**
  - Thread = unit of dispatching/execution
  - Process = unit of resource ownership
  - Multithreaded, single-process – all share the same resources, address space & data



  - Benefits of threads:
    - Takes less time to spawn than a process
    - Less time to terminate than a process
    - Less time to switch between threads within the same process
    - More efficient communication between execution entities
    - Threads within the same process share memory
  - Examples of multithreading uses:
    - Foreground + background work
    - Asynchronous processing
    - Speed of execution
    - Modular program structure
- **User-level** vs. **kernel-level** threads**:**
  - ULT – kernel schedules processes; not aware of threads
    - Thread may still be in "running" state when process is blocked
  - KLT – kernel schedules threads
  - Advantages of ULT:
    - Switching threads does not require a mode switch to kernel
    - Scheduling can be application specific
    - Can run on any OS
  - Disadvantages of ULT:
    - ULT making a system call → blocks all threads in the process

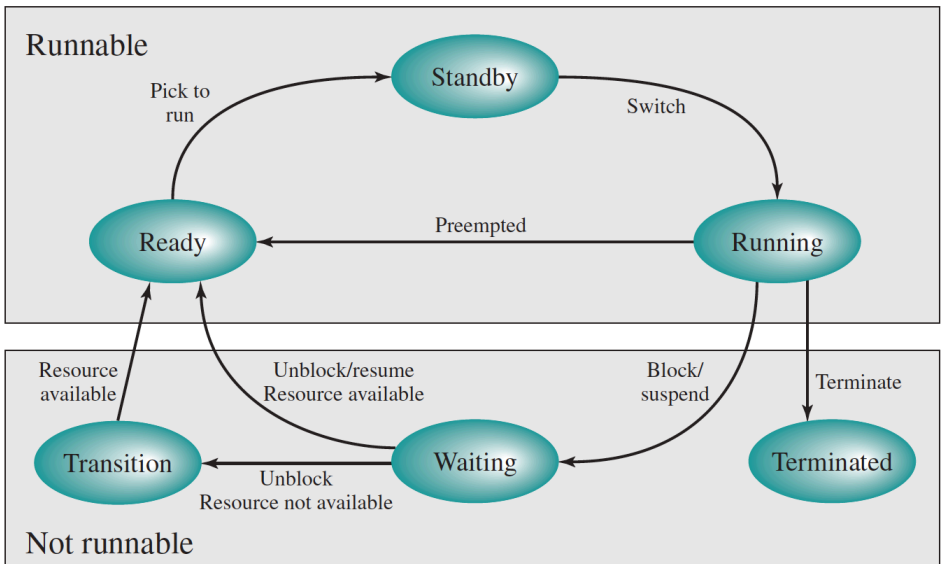
        - Jacketing – wrapper around system call to make it non-blocking to threads
        - In KLT, kernel can schedule another thread of the same process while one is blocked
    - Cannot take advantage of multiprocessing; only 1 thread in a process can execute at a time
        - In KLT, threads of the same process can be run on multiple processors
- Disadvantages of KLT:
    - Thread switch within the same process requires a mode switch (overhead)
- Combined approach – kernel schedules processes & threads



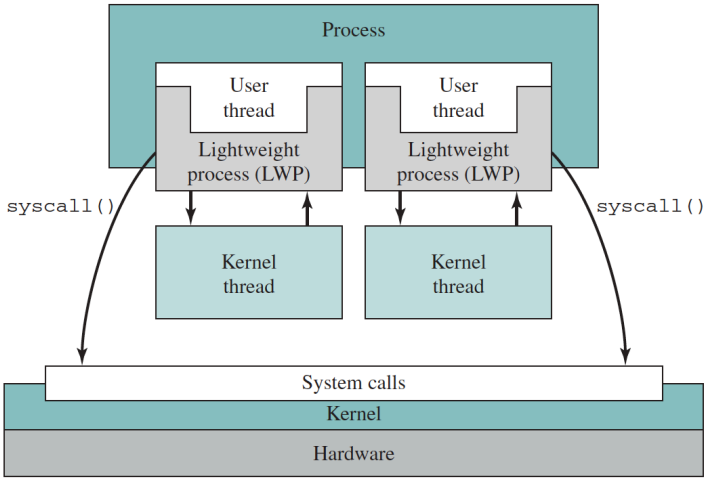
- **Amdahl's Law:**
    - If f = fraction of program that is inherently parallel (without overhead) and $(1 - f)$ is inherently serial, the speedup due to using N processors is

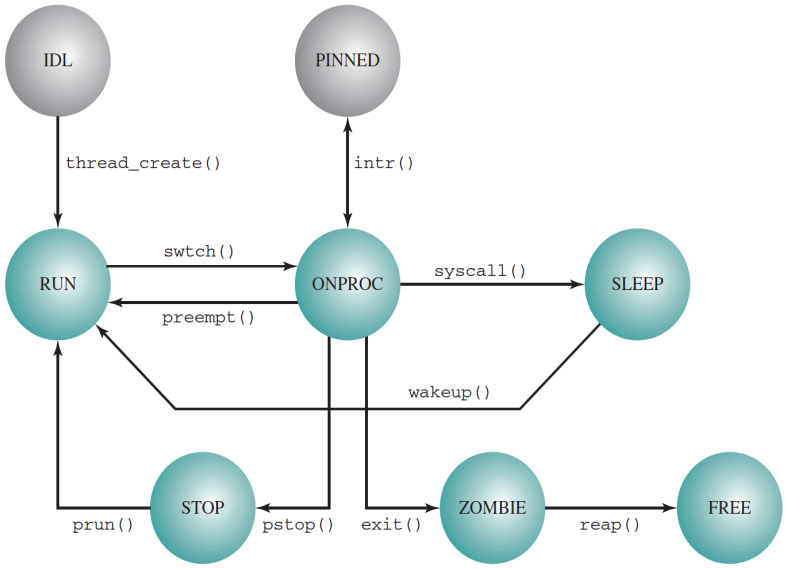
$$\frac{1}{(1-f)+\frac{f}{N}}$$

- **Windows 8**
    - Applications can use user-module scheduling to schedule their own threads

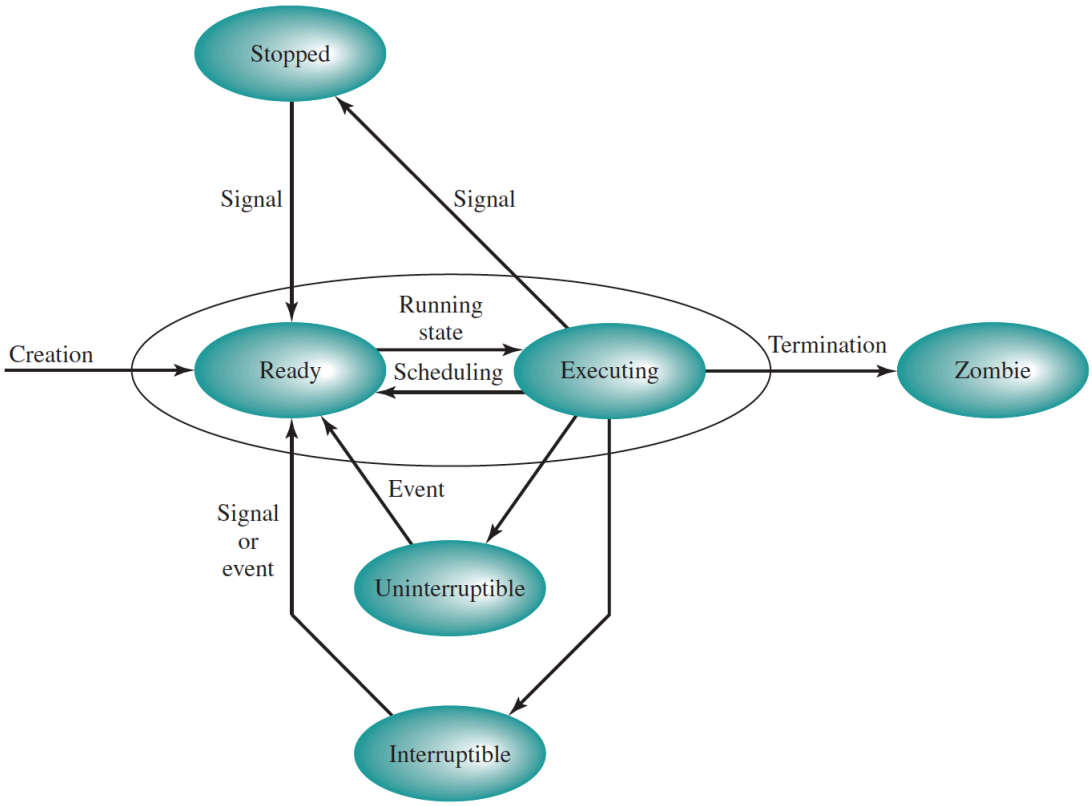


- **Solaris**
  - Lightweight processes – mapping from ULTs to a kernel thread





- **Linux**
  - Namespaces – allow processes to have different views of the systems



Chapter 5 (5.1 – 5.6)

- Definitions:

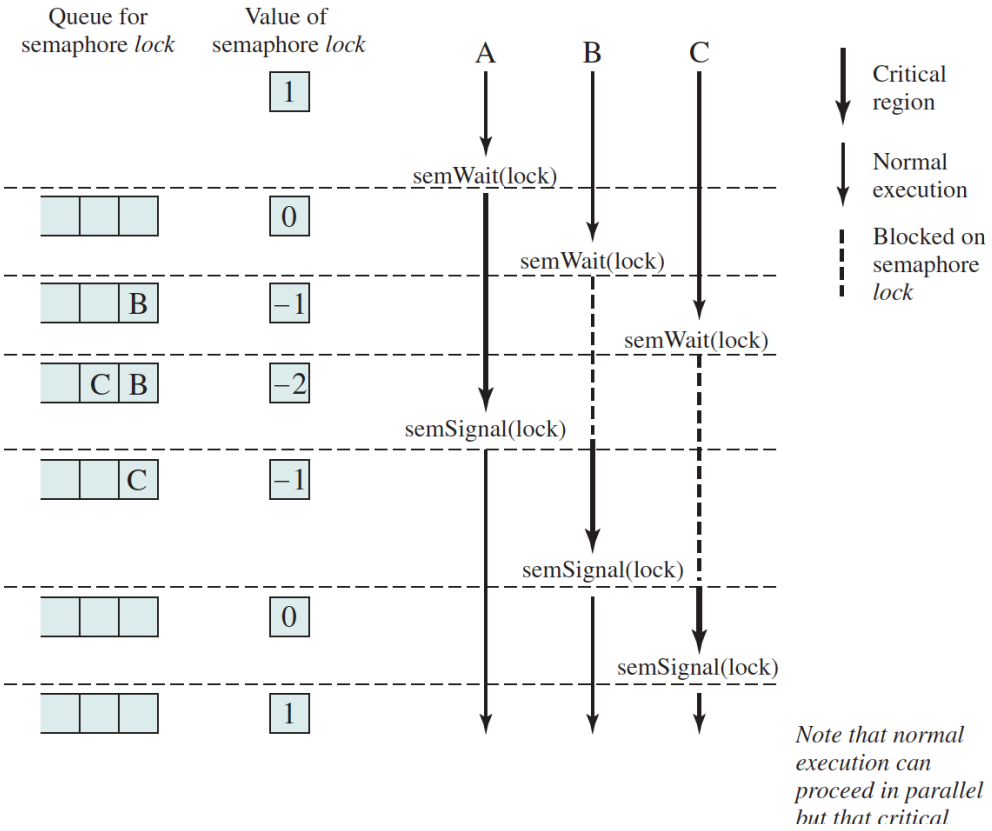
Atomic operation	A function or action implemented as a sequence of one or more instructions that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation. The sequence of instruction is guaranteed to execute as a group, or not execute at all, having no visible effect on system state. Atomicity guarantees isolation from concurrent processes.
Critical section	A section of code within a process that requires access to shared resources and that must not be executed while another process is in a corresponding section of code.
Deadlock	A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.
Livelock	A situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work.
Mutual exclusion	The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.
Race condition	A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.
Starvation	A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.

- Process interaction:

Degree of Awareness	Relationship	Influence that One Process Has on the Other	Potential Control Problems
Processes unaware of each other	Competition	• Results of one process independent of the action of others • Timing of process may be affected	• Mutual exclusion • Deadlock (renewable resource) • Starvation
Processes indirectly aware of each other (e.g., shared object)	Cooperation by sharing	• Results of one process may depend on information obtained from others • Timing of process may be affected	• Mutual exclusion • Deadlock (renewable resource) • Starvation • Data coherence
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	• Results of one process may depend on information obtained from others • Timing of process may be affected	• Deadlock (consumable resource) • Starvation

- Requirements for **mutual exclusion**:
  - Enforcing that only one process at a time is allowed into its *critical section* (CS)
  - A process must be able to halt in a non-CS without interfering w/ other processes
  - No deadlock or starvation
  - A process must be able to enter its CS w/o delay when no one else is in it
  - No assumptions made about processor speeds or # of processors

- Process remains inside its CS for a finite time
- Hardware support for mutual exclusion:
  - Interrupt disabling within CS
    - Inefficiency – can't interleave processes
    - Doesn't work for multiprocessor
  - Special machine instructions – designed to be atomic while executing
    - E.g. restricts access to the same memory location during execution
    - Uses a "bolt" (flag) that is set/unset to indicate whether CS is occupied or not
    - Advantages:
      - Applicable to any # of processors
      - Can support multiple CSs
    - Disadvantages:
      - Busy waiting – consumes processor time
      - Deadlock and starvation possible
- **Semaphores**
  - Semaphore – an integer with 3 possible operations:
    - Initialize to a value ≥ 0
    - Wait – decrements value
      - If value < 0, process is blocked
    - Signal – increments value
      - If value ≤ 0, unblock a process
    - At any time:
      - If value ≥ 0 → # of processes that can enter CSs without being blocked
      - If value < 0 → # of processes blocked



    - Binary semaphore
        - Initialize to 0 or 1
        - Wait – set to 0
            - If value was already 0, process is blocked
        - Signal – if there are blocked processes, unblock one
            - Else set value to 1
    - Mutex – binary semaphore
        - Process that locks it must be the one to unlock it
    - Call wait before entering CS, call signal after
- **Monitors**
    - Software module that "wraps" a CS
    - Characteristics:
        - Local data variables are only accessible by monitor's procedures
        - A process enters the monitor by invoking one of its procedures
        - Only 1 process can execute in the monitor at a time
- **Message passing**
    - Blocking send + receive (rendezvous)
        - Tight synchronization b/t processes
    - Non-blocking send + blocking receive
        - Most useful; sender can send many messages
    - Non-blocking send + receive
        - Messages might get lost, if receive is called too early
    - Direct addressing – specify receiving process id
        - Receiver could know sender ahead of time
    - Indirect addressing (mailbox) – shared queues that hold messages to be picked up
    - Messages can be used to enforce mutual exclusion:
        - One message in shared mailbox
        - Whoever receives the message is in CS; others on blocked on receive
- Examples of mutual exclusion in practice:
    - Producer/consumer problem
        - Multiple producers placing data in a buffer
        - One consumer reading data from buffer
        - Only one can access buffer at once
    - Reader/writers problem
        - Multiple readers may read the file
        - One writer at a time may write to the file
        - If a writer is writing to the file, no reader may read it

Chapter 6 (6.1 – 6.8)

-