Processes and Threads

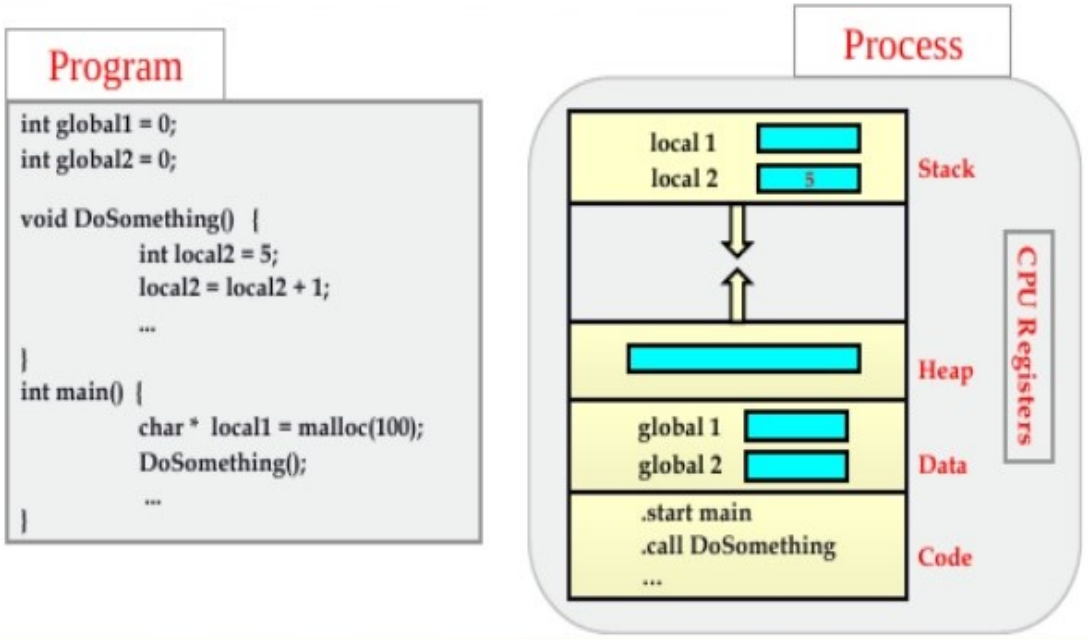
Operating Systems

Processes

- A process is an abstraction of a running program
- It enables doing several things at the same time
  - support the ability to have (pseudo) concurrent operation
  - Ex. consider a user PC ??
- In a multiprogramming system, the CPU switches from process to process quickly, running each for tens or hundreds of milliseconds

Process and Program

- A process is an activity of some kind
- A program is something that may be stored on disk, not doing anything
- Ex: baking vs. recipe

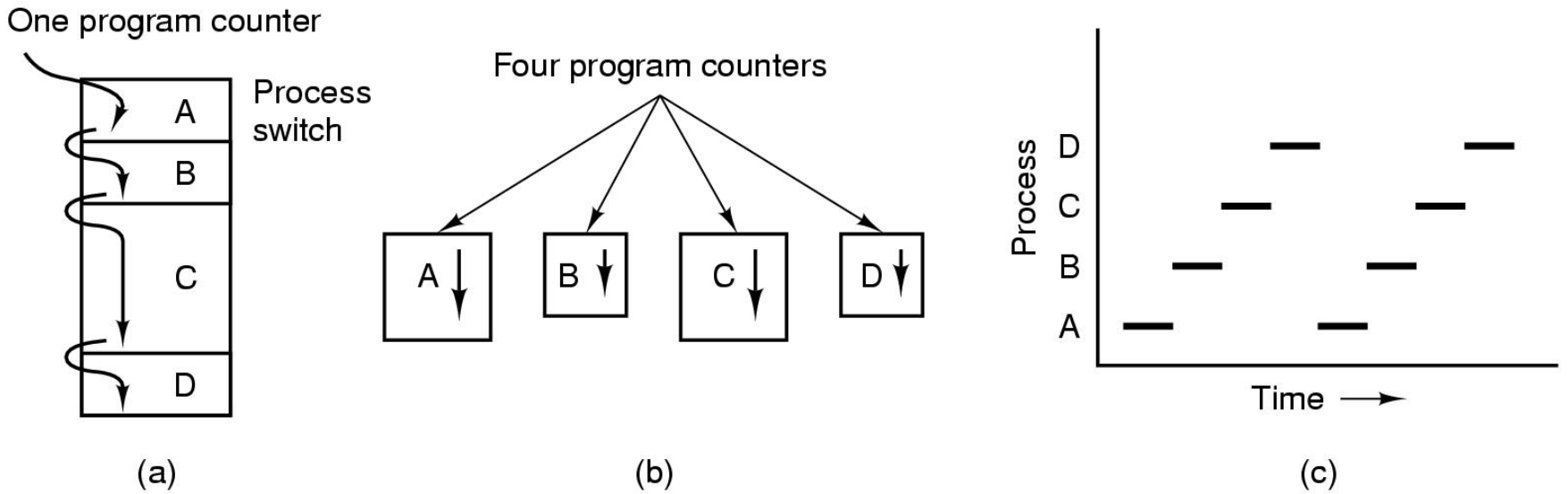


The Process Model

- All the runnable software on the computer, sometimes including the operating system, is organized into a number of **sequential processes**
- To understand the system, it is much easier to think about a collection of processes running in (pseudo) parallel
- This rapid switching back and forth is called **multiprogramming**

The Process Model

a) Multiprogramming four programs.
b) Conceptual model of four independent, sequential processes.
c) Only one program is active at once.



(we assume there is only one CPU, simpler to understand)

Process Creation

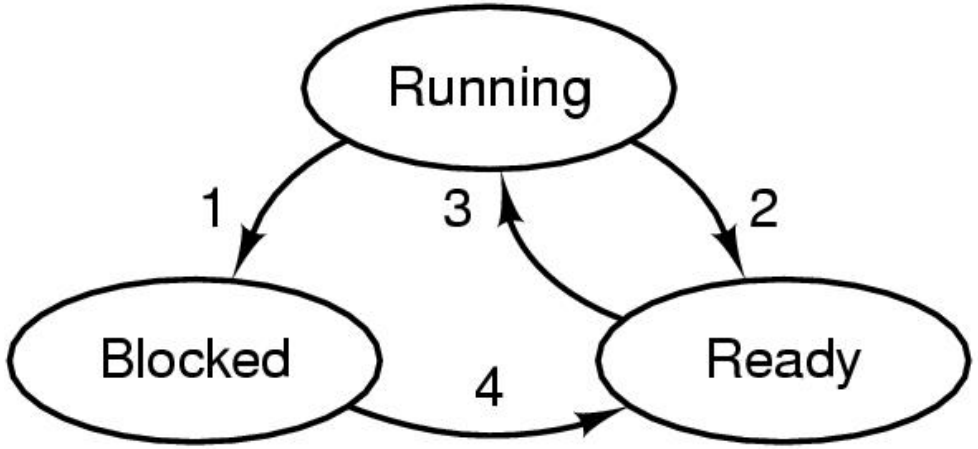
- Events which cause processes creation:
  1. System initialization.
  2. Execution of a process creation system call by a running process.
  3. A user request to create a new process.
  4. Initiation of a batch job.
- In all these cases, a new process is created by having an existing process execute a *process creation system call*.

Process Termination

- Events which cause process termination:
  1. Normal exit (voluntary).
  2. Error exit (voluntary).
  3. Fatal error (involuntary).
  4. Killed by another process (involuntary).

Process States

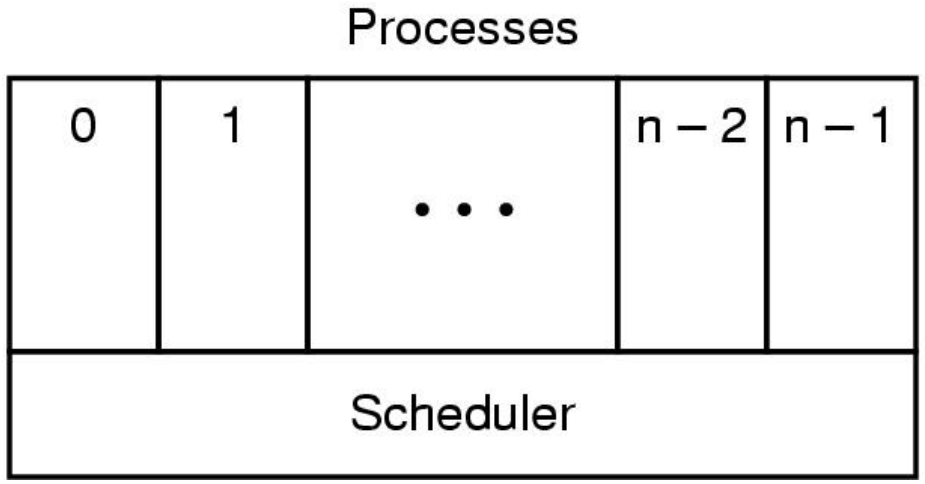
- Three states a process may be in:
  1. Running (actually using the CPU at that instant).
  2. Ready (runnable; temporarily stopped to let another process run).
  3. Blocked (unable to run until some external event happens).



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

Implementation of Processes

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



Process Table

- One entry per process
- Contains important information about the process' state
- Information that must be saved when the process is switched from *running* to *ready* or *blocked* state so that it can be restarted later

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		

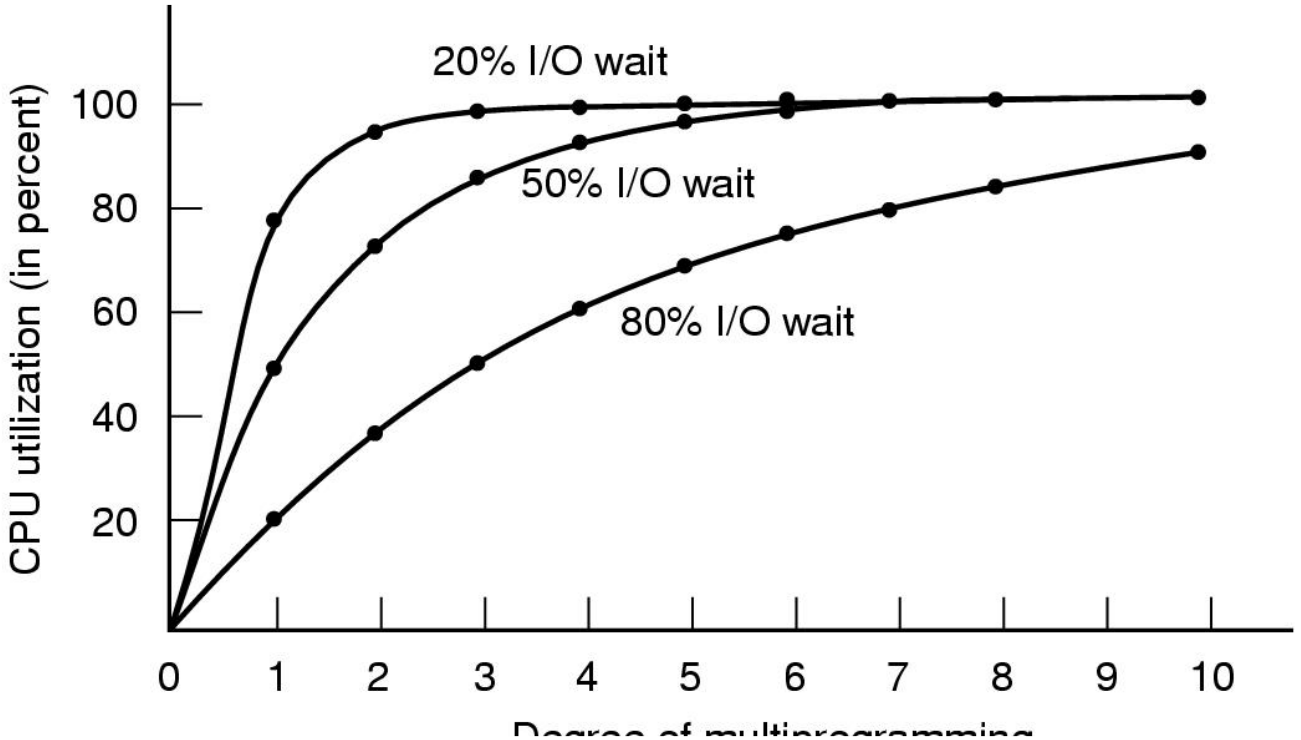
Interrupt Handling and Scheduling

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

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.

Modelling Multiprogramming

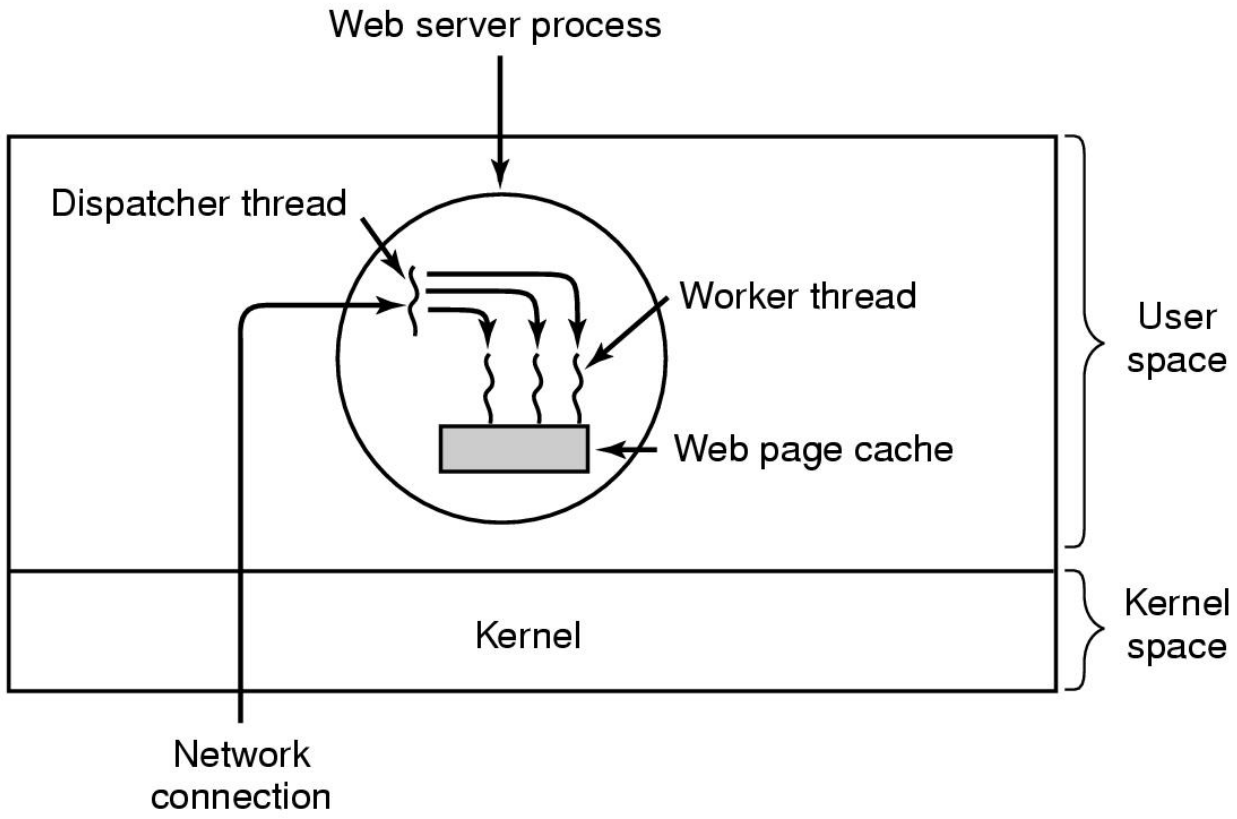
- CPU utilization as a function of the number of processes in memory.



Threads

- In many applications, multiple activities are going on at once
- Threads give the ability for the parallel entities to share an address space and all of its data
  - not possible with multiple processes (with their separate address spaces)
- Threads are lighter weight than processes, so they are easier (i.e., faster) to create and destroy
- Threads allow computing and IO activities to overlap

Thread Usage - A Multithreaded Web Server



- The **dispatcher** reads incoming requests for work from the network and chooses an idle (i.e., blocked) **worker thread** to hand the request

Thread Usage - A Multithreaded Web Server

- A rough outline of the code for
  a) Dispatcher thread.
  b) Worker thread.

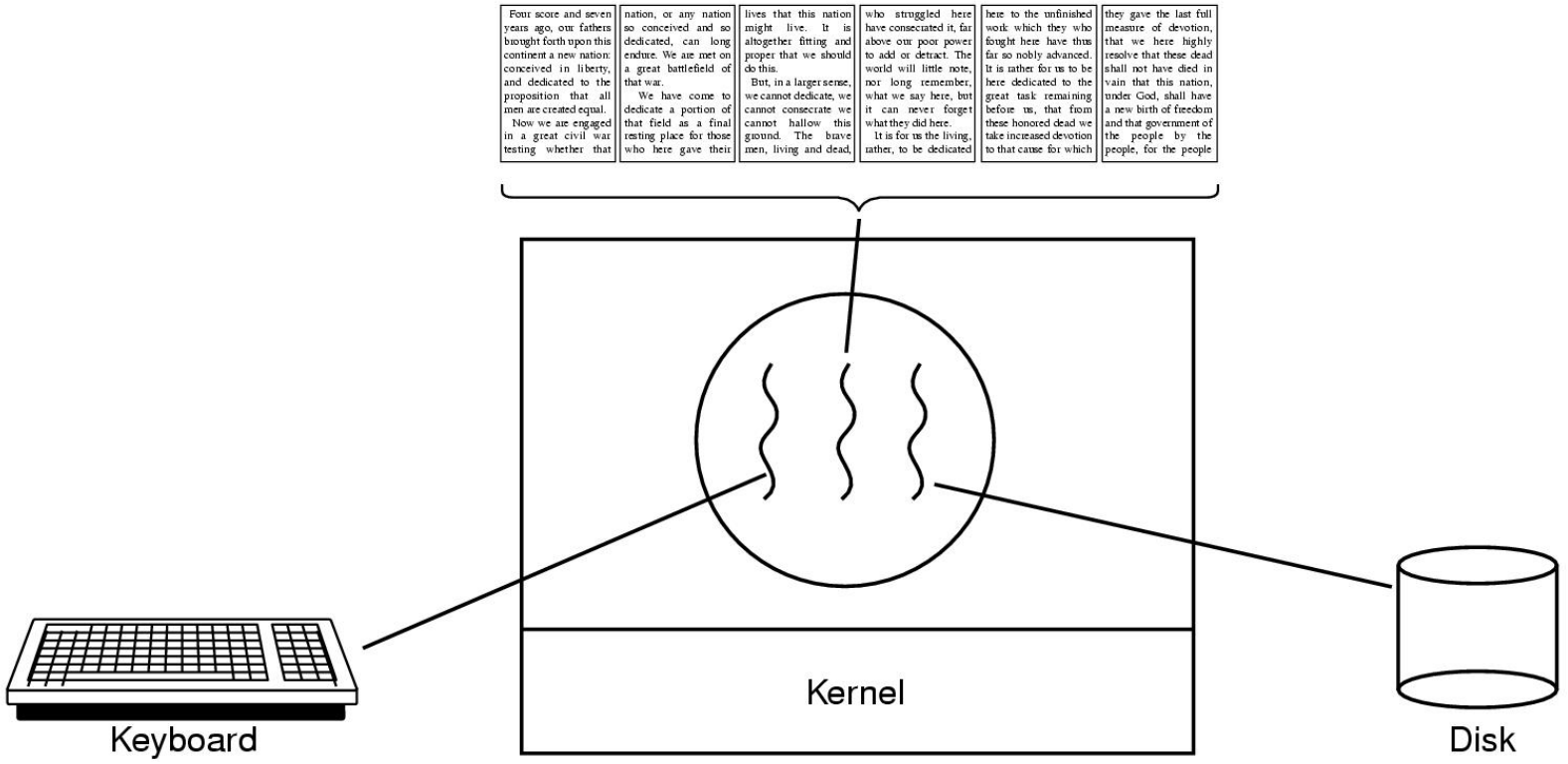
```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

(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)

*Thread Usage - A Word Processor

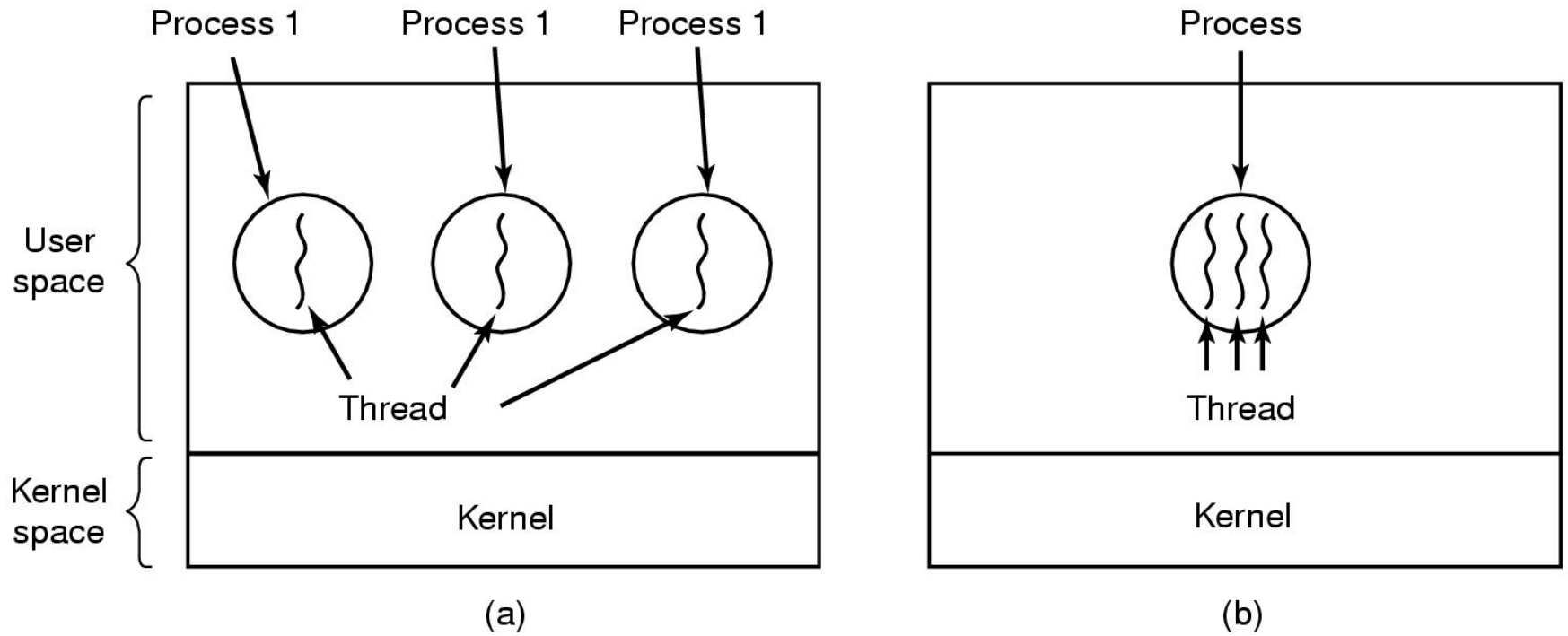


- First thread interacts with the user.
- Second thread handles reformatting in the background.
- Third thread handles the disk backups.

The Classical Thread Model

- **Multithreading** is the situation of allowing multiple threads in the same process

(a) Three processes each with one thread. (b) One process with three threads.



The Classical Thread Model

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

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	

Advantages of Thread over Process

- Responsiveness:
  - If the process is divided into multiple threads, one thread can complete its execution and its output can be immediately returned.
- Faster context switch:
  - Context switch time between threads is lower compared to process context switch. Process context switching needs more CPU overhead.
- Effective utilization of multiprocessor system:
  - If we have multiple threads in a single process, then we can schedule multiple threads on multiple processor.

Advantages of Thread over Process

- Resource sharing:
  - Resources like code, data, and files can be shared among all threads within a process. (Stack and registers can’t be shared)
- Communication:
  - Communication between multiple threads is easier, as the threads shares common address space. while in process we have to follow some specific communication technique for communication between two process.
- Enhanced throughput of the system:
  - If a process is divided into multiple threads, and each thread function is considered as one job, then the number of jobs completed per unit of time is increased, thus increasing the throughput of the system.

Inter-process Communication (IPC)

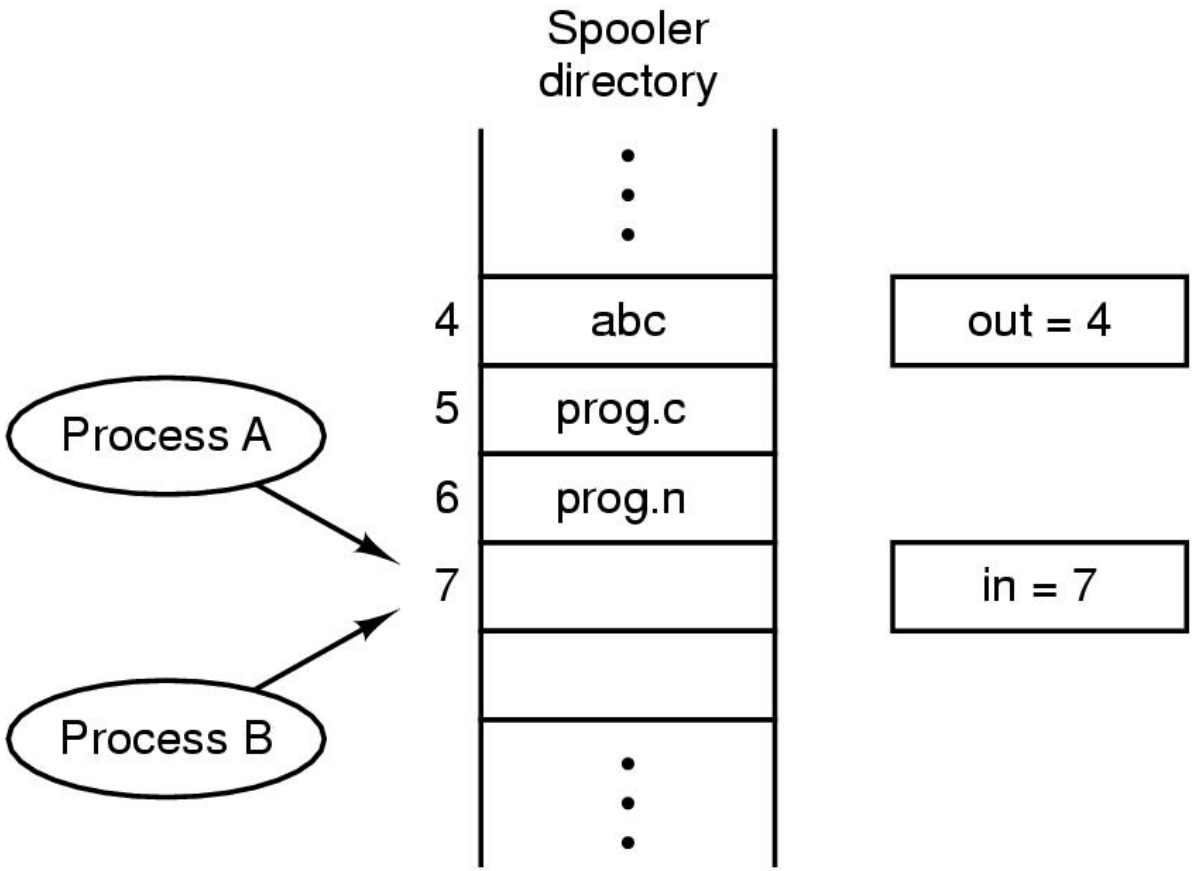
- Processes frequently need to communicate with other processes
  - To pass information from process to another
  - To make sure two or more processes do not get in each other's way
  - To enable proper sequencing when dependencies are present

Mechanisms for IPC

- To ensure that no two processes are ever in their critical regions at the same time.
- Atomic read/write
  - Alternation
- Locks
  - Primitive, minimal semantics, used to build others
- Semaphores
  - Basic, easy to get the hang of, but hard to program with
- Monitors
  - High-level, requires language support, operations implicit
- Message Passing
  - System calls to send and receive messages between processes

Race Conditions

- Two or more processes are reading or writing some shared data and the final result depends on who runs precisely when
- Ex. two processes want to access the printer spooler directory at the same time

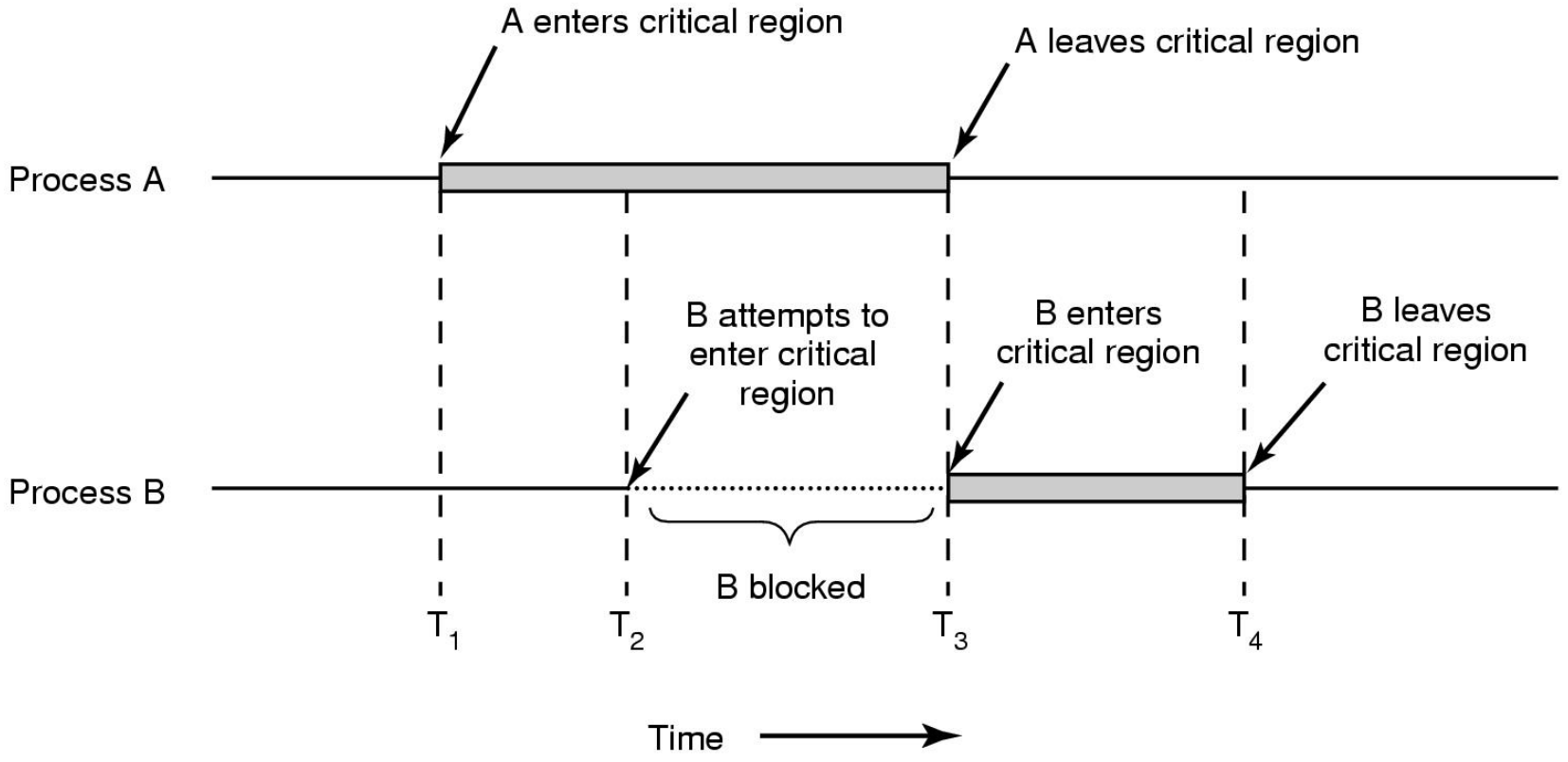


Conditions Required to Avoid Race Condition

1. No two processes may be simultaneously inside their critical regions. – Mutual Exclusion (mutex)
2. No assumptions may be made about speeds or the number of CPUs. – No Assumption
3. No process running outside its critical region may block other processes. – Progress
4. No process should have to wait forever to enter its critical region. – No Starvation

Mutual Exclusion using Critical Regions

- Critical region – the part of the program where shared variables are accessed



Mutual Exclusion with Busy Waiting

- Disabling interrupts
  - Each process disables all interrupts just after entering its critical region and re-enable them just before leaving it
  - In a multicore (i.e., multiprocessor system) disabling the interrupts of one CPU does not prevent other CPUs from interfering with operations the first CPU is performing
- Lock variables
  - A single, shared (lock) variable, process sets it to 1 and enters the critical region. If the lock is already 1, the process just waits until it becomes 0.
  - Has the same fatal flaw that we saw in the spooler directory.

Mutual Exclusion with Busy Waiting

- Strict alternation
  - two processes strictly alternate in entering their critical regions
  - it is not really a serious candidate as a solution because it violates condition 3
- Peterson's solution
- The TSL instruction

Semaphores

- Semaphores are an **abstract data type** that provide mutual exclusion to critical region
- Semaphores are **integers** that support two operations:
  - wait(semaphore): decrement, block until semaphore is open – Down
  - signal(semaphore): increment, allow another thread to enter – Up
- Down : checks the value of the semaphore
  - If value is > 0, then it decrements it (by using one wakeup signal) and continues
  - If value = 0, then the process is put to sleep without completing its down operation
- Up : increments the value of the semaphore
  - If there are processes sleeping on the semaphore, then one of them is chosen, and it is allowed to complete its down operation
- Checking the value of a semaphore and updating them is done in an atomic fashion

Monitors

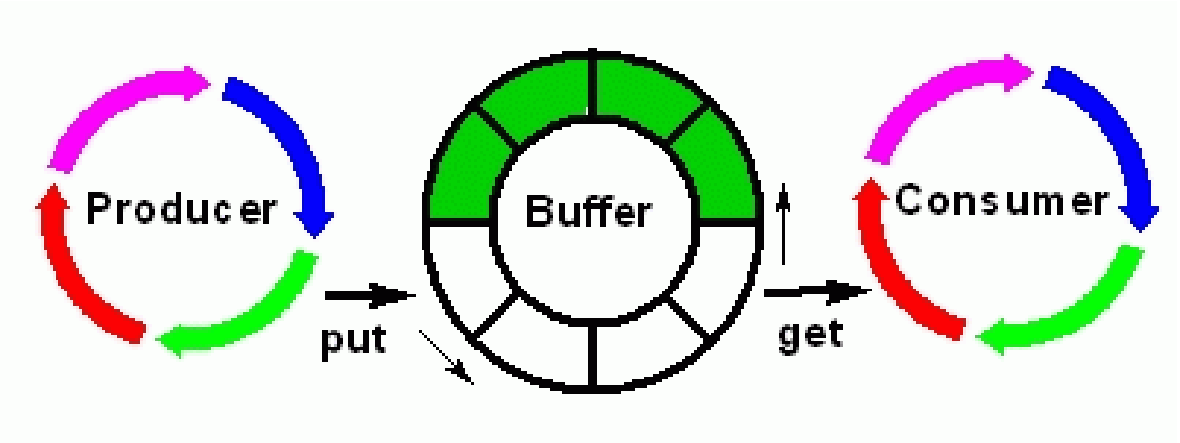
- A monitor is a programming language construct that controls access to shared data
  - Synchronization code added by compiler, enforced at runtime
- A monitor protects its data from unstructured access
  - It guarantees that threads accessing its data through its procedures interact only in legitimate ways
- For ensuring mutual exclusion, programmer turns all the critical regions into monitor procedures, this way no two process can enter their critical region at the same time
- When a process calls a monitor procedure:
  - The first couple of instructions of the procedure will check to see if any other process is currently using the monitor
  - If so the calling process will suspended until the other process leaves the monitor
  - If no process is using the monitor, the calling process may enter

Message Passing

- Uses two system calls, *send* and *receive*
  - send(destination, &message);
  - receive(source, &message);
- Simple model of communication and synchronization based on atomic transfer of data across a channel
- Commonly used in parallel programming systems
- Direct application to distributed systems

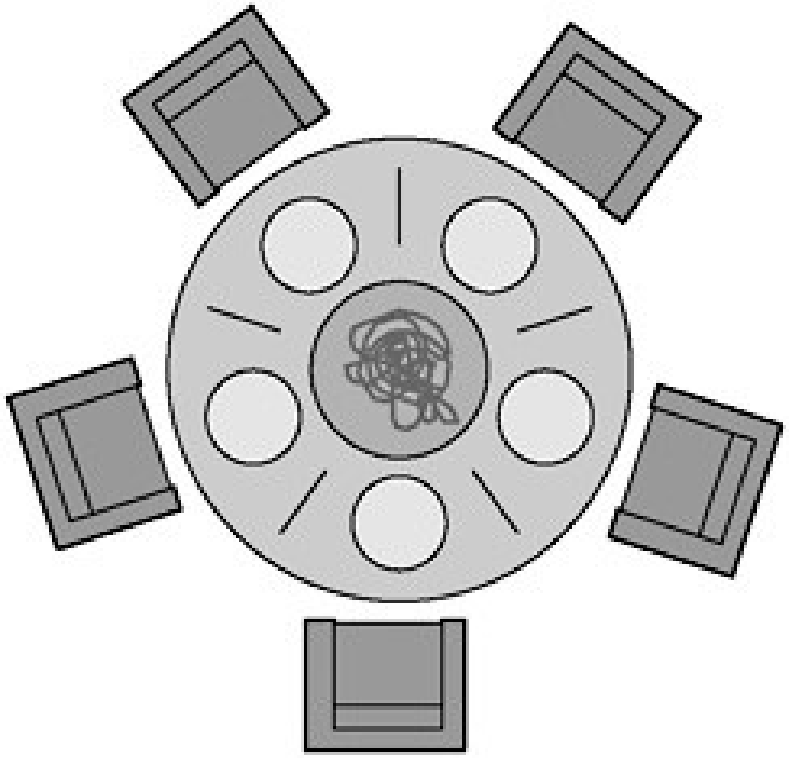
Classical IPC Problems

- Producer–consumer problem
  - Models access to a bounded-buffer
  - Producer won't try to add data into the buffer if it's full
  - Consumer won't try to remove data from an empty buffer



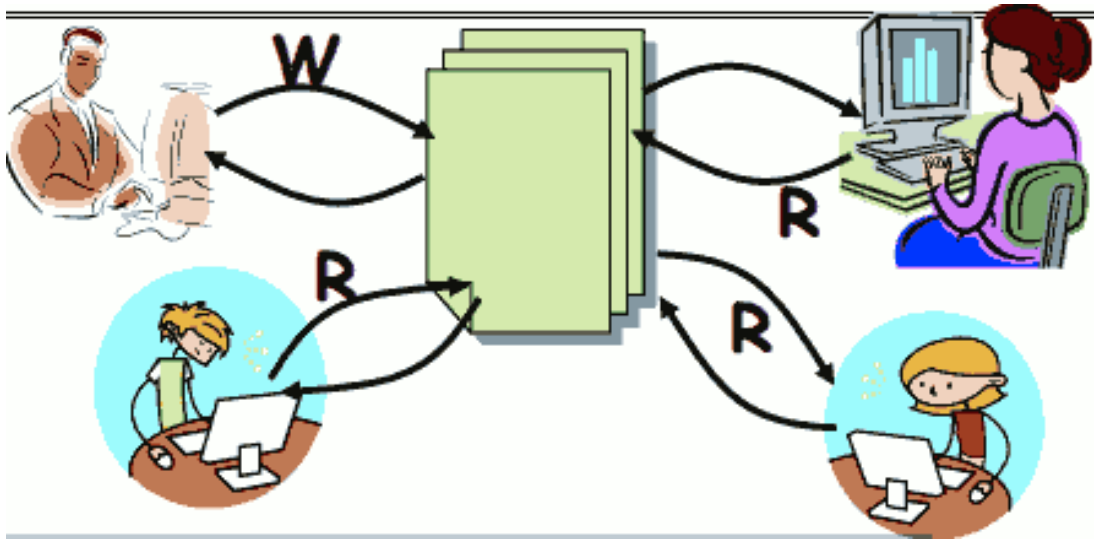
Classical IPC Problems

- Dining philosophers problem (Dijkstra)
  - Models processes competing for exclusive access to a limited number of resources such as I/O devices



*Classical IPC Problems

- Readers–writers problem
  - Models access to a database
  - Two readers can read at once
  - A writer should not wait longer than needed
  - Fairness for both readers and writers



Process Summary

- What are the units of execution?
  - Processes
- How are those units of execution represented?
  - Process Control Blocks (PCBs)
- How is work scheduled in the CPU?
  - Process states, process queues, context switches
- What are the possible execution states of a process?
  - Running, ready, waiting
- How does a process move from one state to another?
  - Scheduling, I/O, creation, termination
- How are processes created?
  - CreateProcess (Windows), fork/exec (Unix)

Threads Summary

- The operating system as a large multithreaded program
  - Each process executes as a thread within the OS
- Multithreading is very useful for applications
  - Efficient multithreading requires fast primitives
  - Processes are too heavyweight
- Solution is to separate threads from processes
  - Kernel-level threads much better, but still significant overhead
  - User-level threads even better, but not well integrated with OS