Fundamentals of the operating systems

Visteon Engineering Academy IV, Sofia, 2018

Lyubomir Tsirov, *ltsirov@visteon.com*

Visteon Confidential

Visteon®

Contents

Visteon

1.What is an operating system?

2.Kernel

3.System call

4.Concurrency – issues and solutions

5.Multi process programming

6.Multi thread programming

7.Advanced OS concepts

- Scheduling(time sharing/time slicing)
- Memory management & Virtual memory(address space)
- I/O access
- Interrupts
- Real time operations

8.Inter process communication in UNIX

ltsirov@visteon.com

What is an operating system?

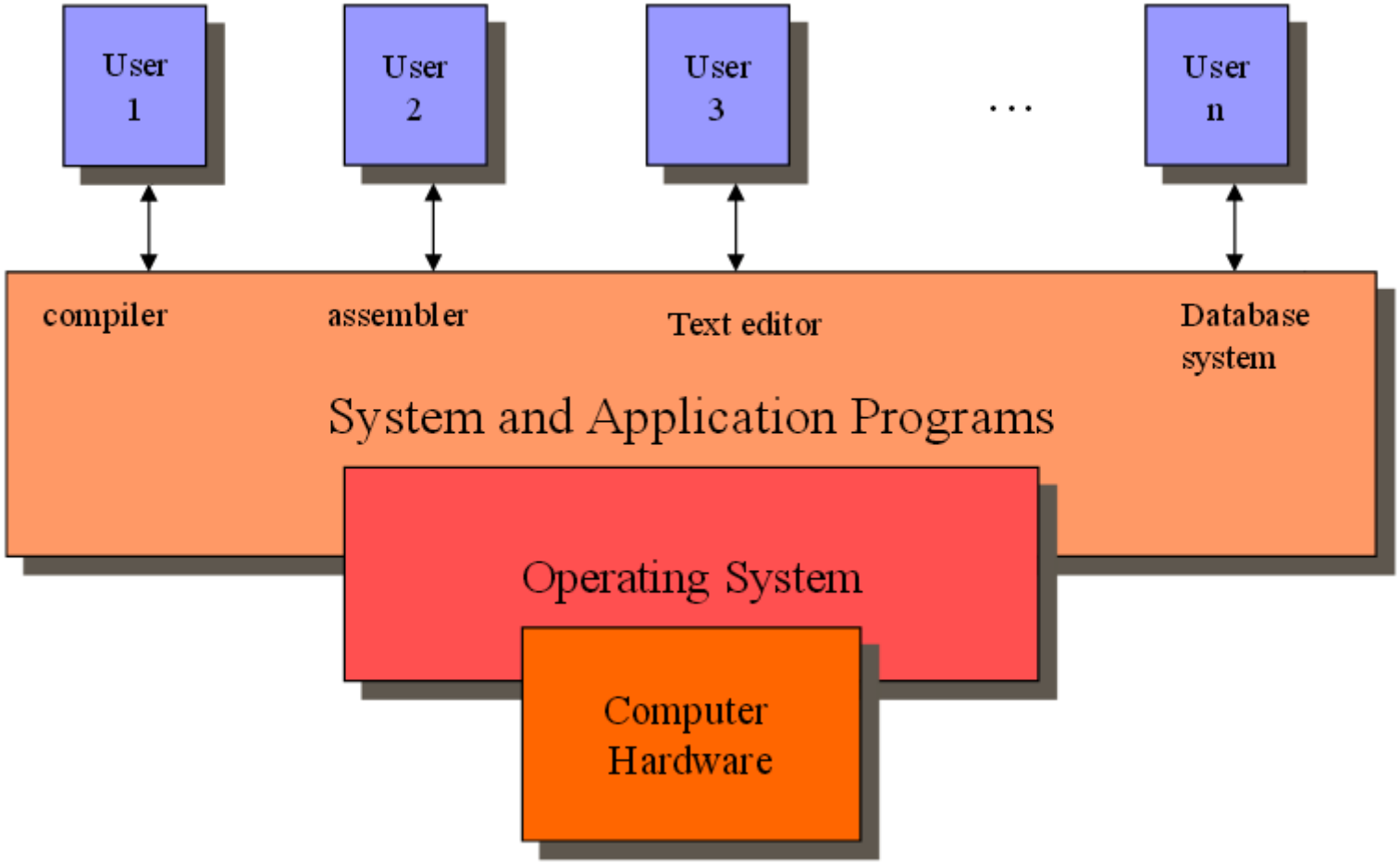
Visteon

A system software/collection of procedures that:

- manage all the system's hardware resources
- control user actions to prevent errors and improper system usage
- provide the users the environment in which they can:
  - use the system resources
  - run their own applications

ltsirov@visteon.com

What is an operating system?



ltsirov@visteon.com

Operating system: Definitions

Visteon

- Resource allocator
  - to allocate resources (software and hardware) of the computer system and manage them efficiently.
- Control program
  - controls the execution of user programs and operations of I/O devices.
- Kernel
  - the one program running at all time. Everything else is either a system program (ships with the operating system) or an application program.

ltsirov@visteon.com

Contents

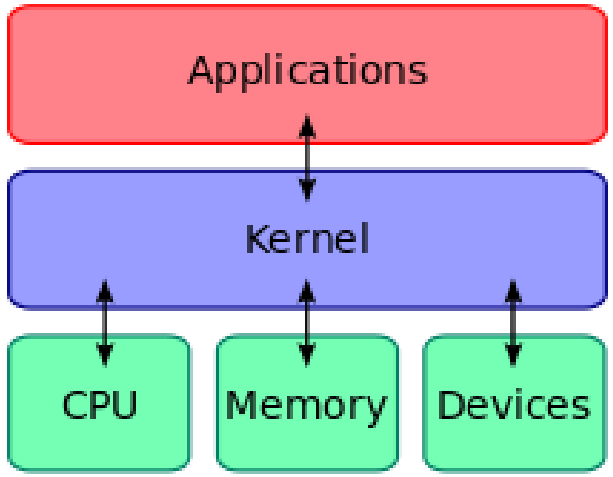
1. What is an operating system?
2. **Kernel**
3. System call
4. Concurrency – issues and solutions
5. Multi process programming
6. Multi thread programming
7. Advanced OS concepts
   - Scheduling(time sharing/time slicing)
   - Memory management & Virtual memory(address space)
   - I/O access
   - Interrupts
   - Real time operations
8. Inter process communication in UNIX

ltsirov@visteon.com

Visteon

What is a kernel?

- The kernel is a fundamental part of any operating system.
- The kernel manages I/O requests from software, and translates them into data processing instructions for the central processing unit and other electronic components of a computer.



ltsirov@visteon.com

Kernel types

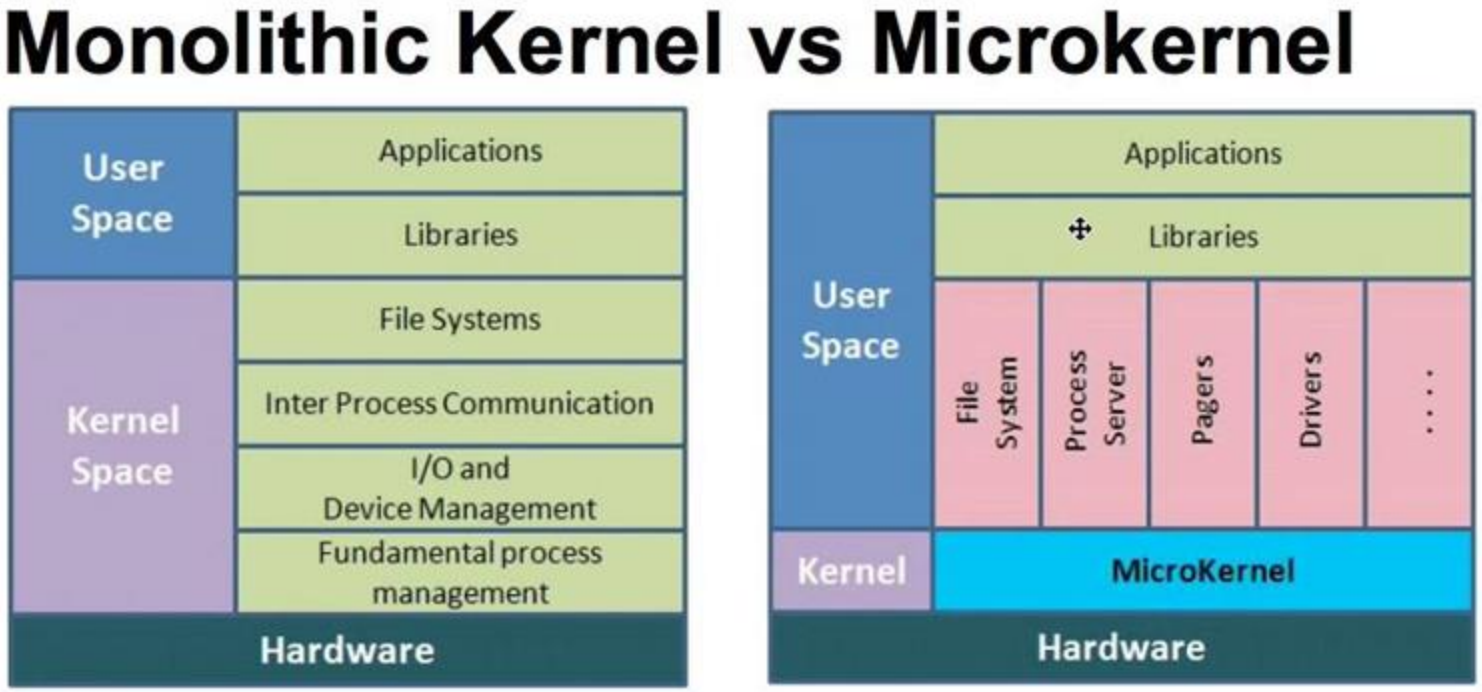
Visteon

- Micro-kernels
  - the kernel modules run in user mode → protection against bugs
  - adaptability to use in distributed systems
  - forces the programmers to adopt a modularize approach
  - easily ported to other architectures
  - better use of RAM than monolithic kernels
- Monolithic kernels
  - monolithic OS faster than micro-kernel OS
  - modularized approach
  - platform independence
  - frugal main memory usage
  - no performance penalty

ltsirov@visteon.com

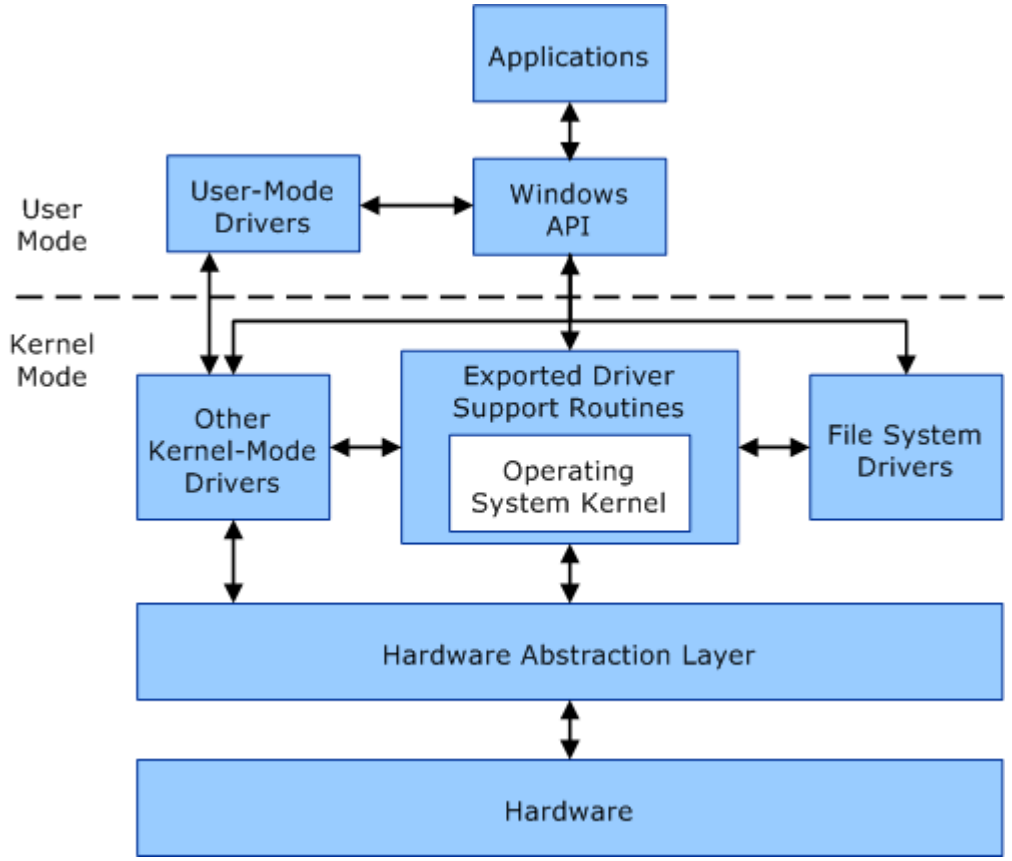
Kernel types

Visteon



ltsirov@visteon.com

Kernel mode vs User mode



- Main reason for separation of kernel and user space is **security!**

ltsirov@visteon.com

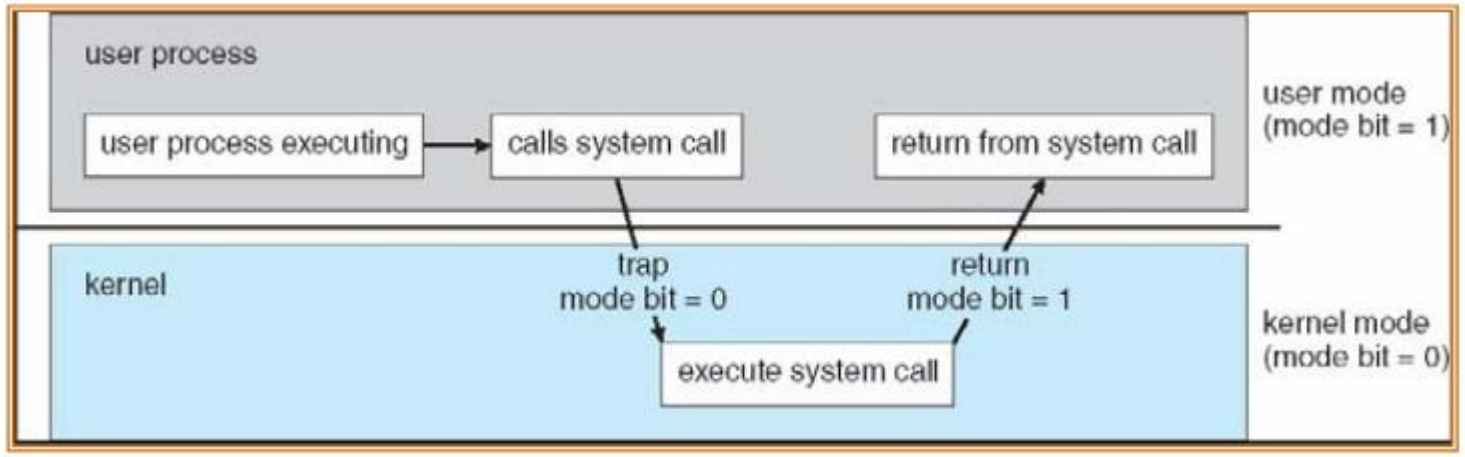
Contents

1. What is an operating system?
2. Kernel
3. **System call**
4. Concurrency – issues and solutions
5. Multi process programming
6. Multi thread programming
7. Advanced OS concepts
   - Scheduling(time sharing/time slicing)
   - Memory management & Virtual memory(address space)
   - I/O access
   - Interrupts
   - Real time operations
8. Inter process communication in UNIX

ltsirov@visteon.com

Visteon

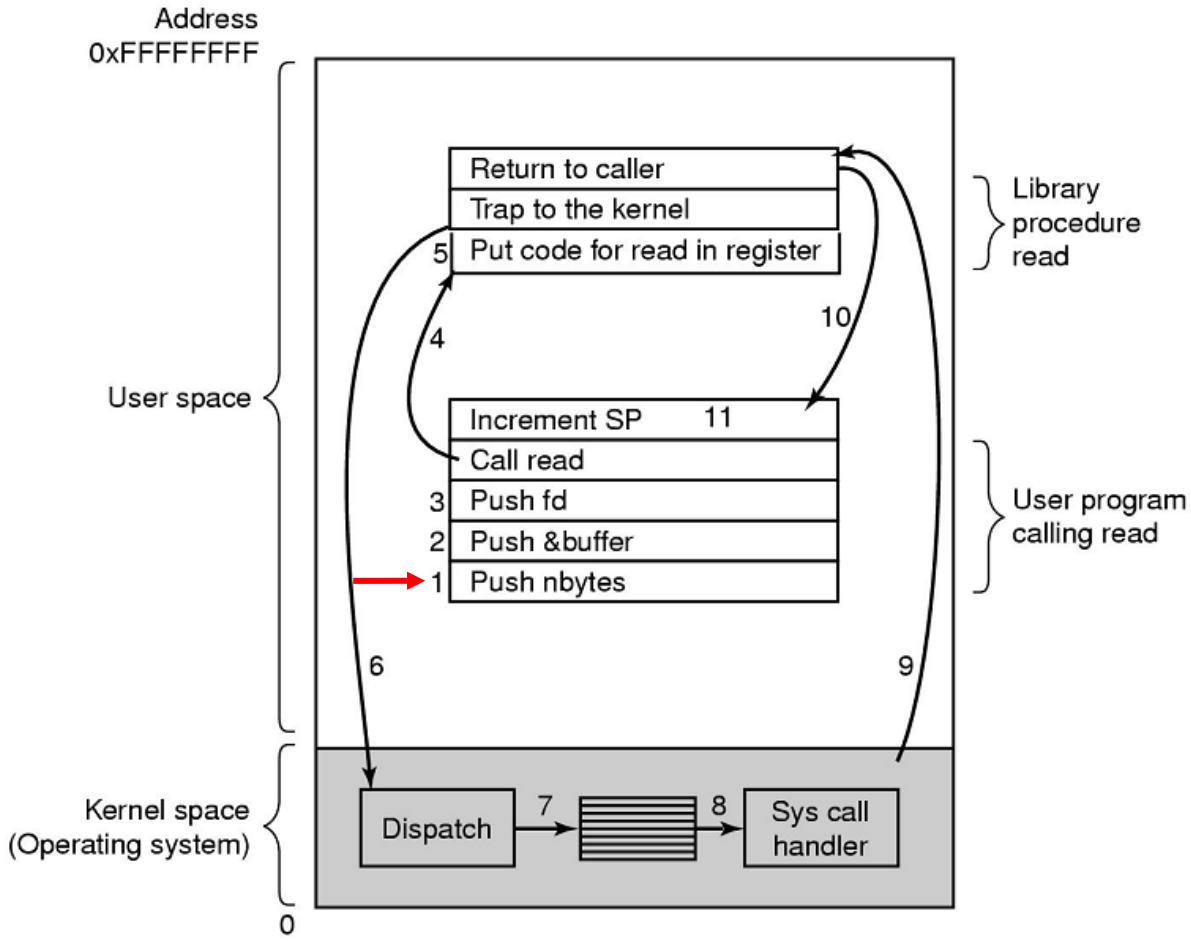
- Definition
  - a call to an OS service
  - a trap into the OS code
- Examples of system calls
  - File manipulation: open(), read(), write(), lseek(), close() …
  - File system management: mkdir(), mount(), link(), chown() …
  - Process management: fork(), exec(), wait(), exit() …



ltsirov@visteon.com

System functions – libc.so

- Steps in making a system call



There are 11 steps executing the system call:
read (fd, buffer, nbytes)

ltsirov@visteon.com

Contents

Visteon

1. What is an operating system?
2. Kernel
3. System call
4. **Concurrency – issues and solutions**
5. Multi process programming
6. Multi thread programming
7. Advanced OS concepts
   - Scheduling(time sharing/time slicing)
   - Memory management & Virtual memory(address space)
   - I/O access
   - Interrupts
   - Real time operations
8. Inter process communication in UNIX

ltsirov@visteon.com

Basic concepts of OS: Concurrency

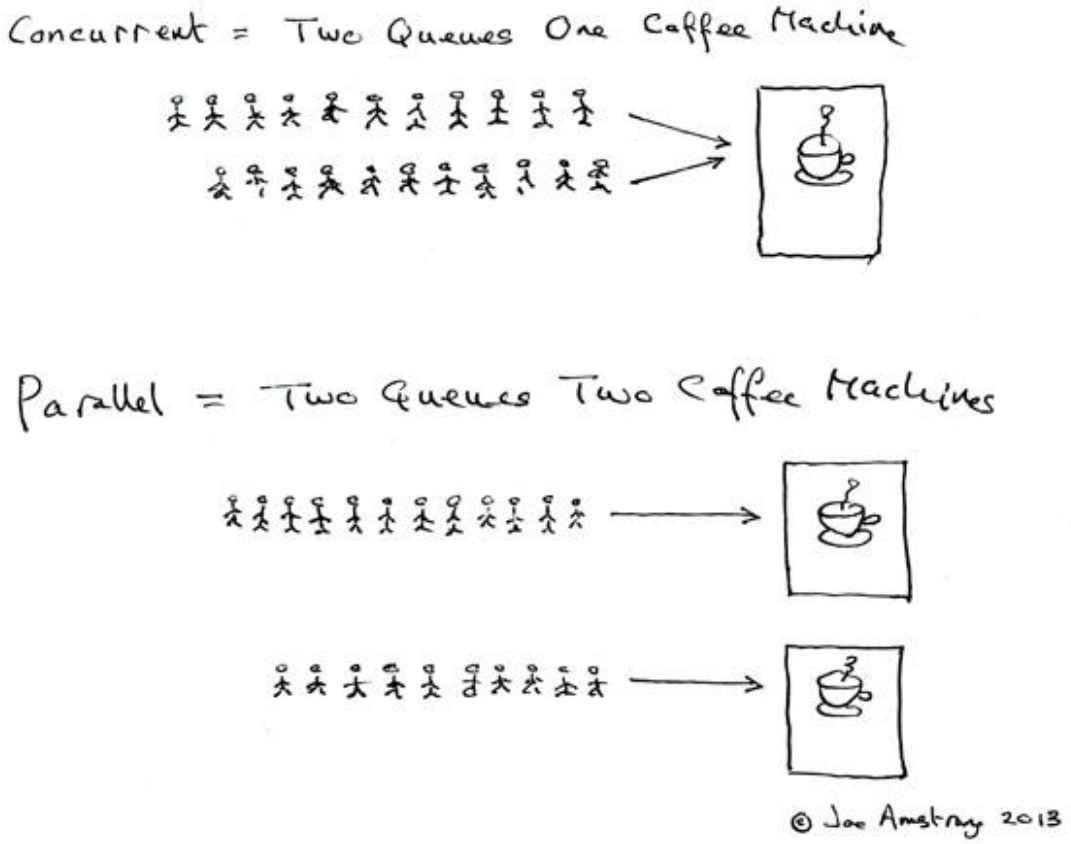
The operating system provides the ability to have more than one independent executions at a given time.

Usually there are no limitations on the number of independent executions at a given time, however, if the separate parallel executions, have to access a single resource, a concurrency arises.

ltsirov@visteon.com

Visteon

Concurrency is the interleaving of processes in time to give the appearance of simultaneous execution. Thus it differs from parallelism, which offers genuine simultaneous execution.



ltsirov@visteon.com

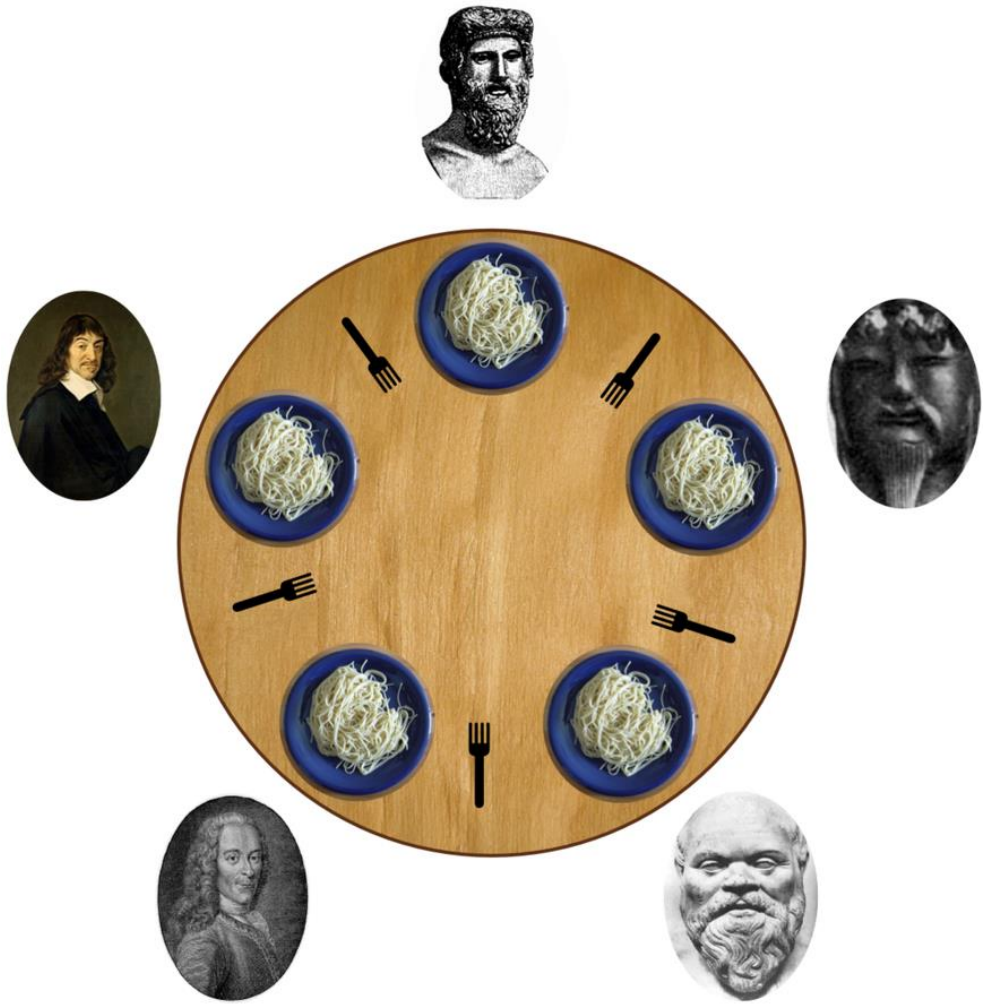
Visteon

Concurrency vs. parallelism

- Concurrency is about dealing with lots of things at once.
- Parallelism is about doing lots of things at once.
- Not the same, but related.
- Concurrency is about structure, parallelism is about execution.
- Concurrency provides a way to structure a solution to solve a problem that may (but not necessarily) be parallelizable.

ltsirov@visteon.com

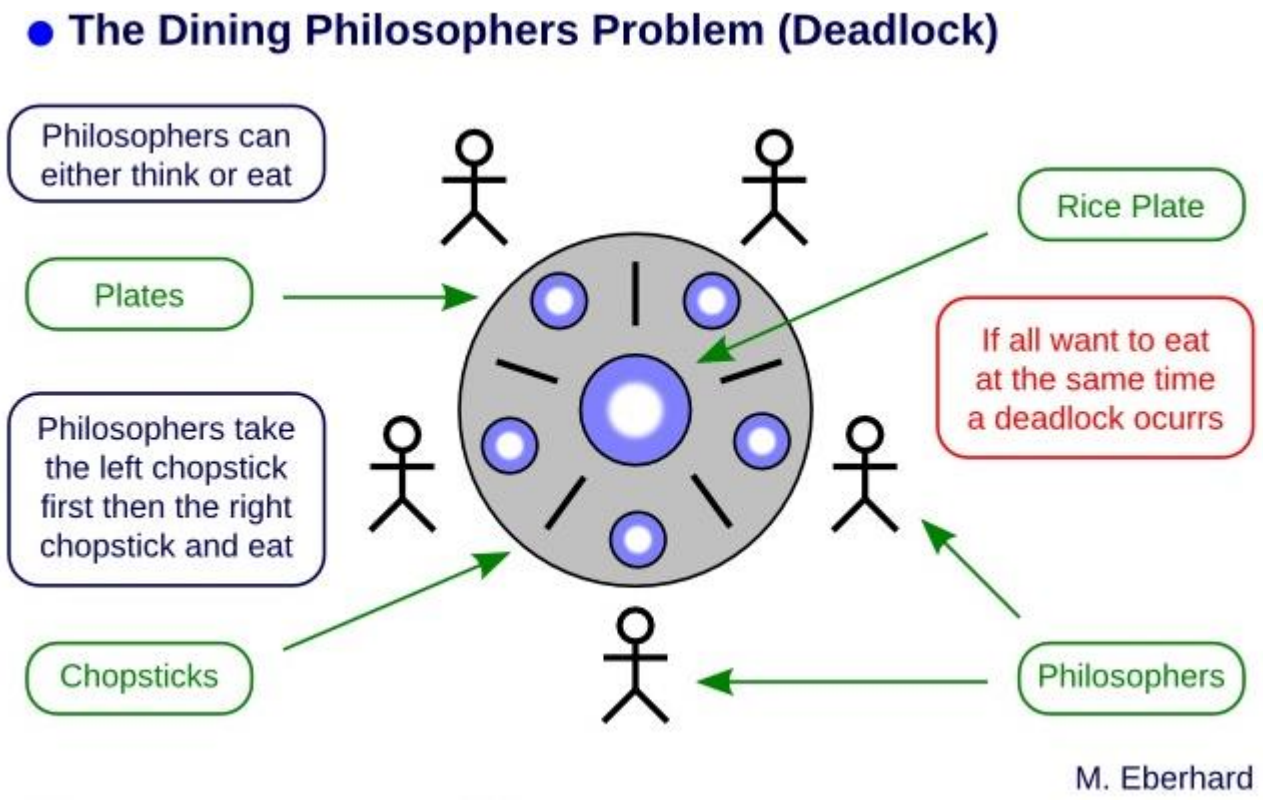
Basic concepts of OS: Concurrency



"Dining philosophers" by Benjamin D. Esham / Wikimedia Commons. Licensed under CC BY-SA 3.0

ltsirov@visteon.com

Basic concepts of OS: Concurrency



ltsirov@visteon.com

Visteon

Concurrency cannot be avoided because:

- Users are concurrent - a person can handle several tasks at once (have you every listened to music while doing other work and heard the phone ring?) and expects the same from a computer.
- Multiprocessors are becoming more prevalent. The Internet is perhaps a huge multiprocessor.
- A distributed system (client/server system) is naturally concurrent.
- A windowing system is naturally concurrent.
- I/O is often slow because it involves slow devices such as disks, printers; many network operations are essentially (slow) I/O operations. When doing I/O it is helpful to handle the I/O concurrently with other work.

ltsirov@visteon.com

Visteon

Concurrency issues:

- Separable operation (non-Atomic)

An operation is atomic if the steps are done as a unit. Operations that are not atomic, but interruptible and done by multiple processes can cause problems.

- Race condition

- A race condition occurs if the outcome depends on which of several processes gets to a point first.

- Situations like this where processes access the same data concurrently and the outcome of execution depends on the particular order in which the access takes place

For example, fork( ) can generate a race condition if the result depends on whether the parent or the child process runs first. Other race conditions can occur if two processes are updating a global variable.

- Blocking and starvation

While neither of these problems is unique to concurrent processes, their effects must be carefully considered. Processes can block waiting for resources. A process could be blocked for a long period of time waiting for input from a terminal. If the process is required to periodically update some data, this would be very undesirable. Starvation occurs when a process does not obtain sufficient CPU time to make meaniful progress.

- Deadlock

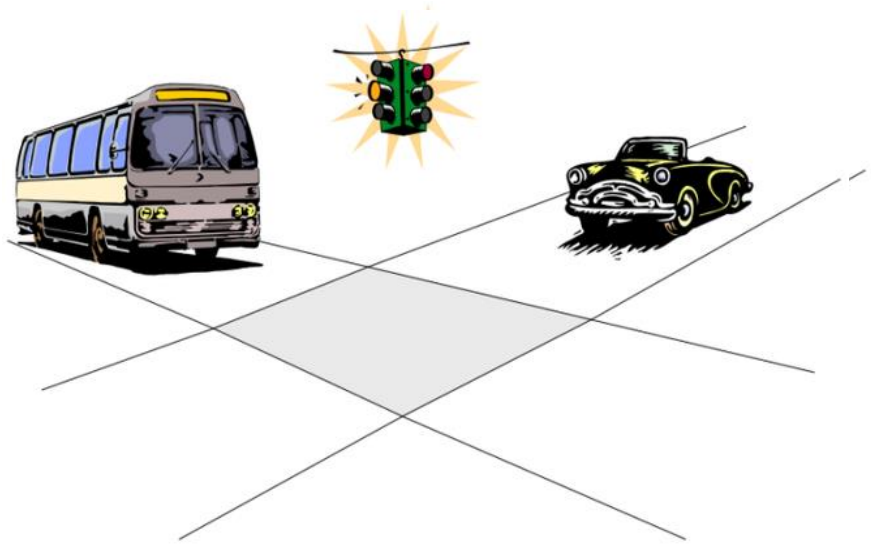
Deadlock occurs when two processes are blocked in such a way that neither can proceed. The typical occurrence is where two processes need two non-shareable resources to proceed but one process has acquired one resource and the other has acquired the other resource. Acquiring resources in a specific order can resolve some deadlocks.

ltsirov@visteon.com

Visteon

Critical section & Mutual exclusion

- When a process executes code that manipulates shared data (or resource), we say that the process is in it's critical section (for that shared data)
- The execution of critical sections must be mutually exclusive: at any time, only one process is allowed to execute in its critical section (even with multiple CPUs)



ltsirov@visteon.com

Visteon

Critical section & Mutual exclusion

- Entry of critical section
  - Each process must request the permission to enter it's critical section (CS)
  - Once entered, each process must lock the critical section, so other cannot be permitted to enter

- Exit of critical section
  - Each process must unlock the critical section so other could be permitted to enter

```
repeat
   entry section
    critical section
   exit section
    remainder section
forever
```

ltsirov@visteon.com

Visteon

Semaphore

- Semaphores are a kind of generalized lock
  - First defined by the Dutch Edsger Dijkstra in late 60s
  - Main synchronization primitive used in original UNIX

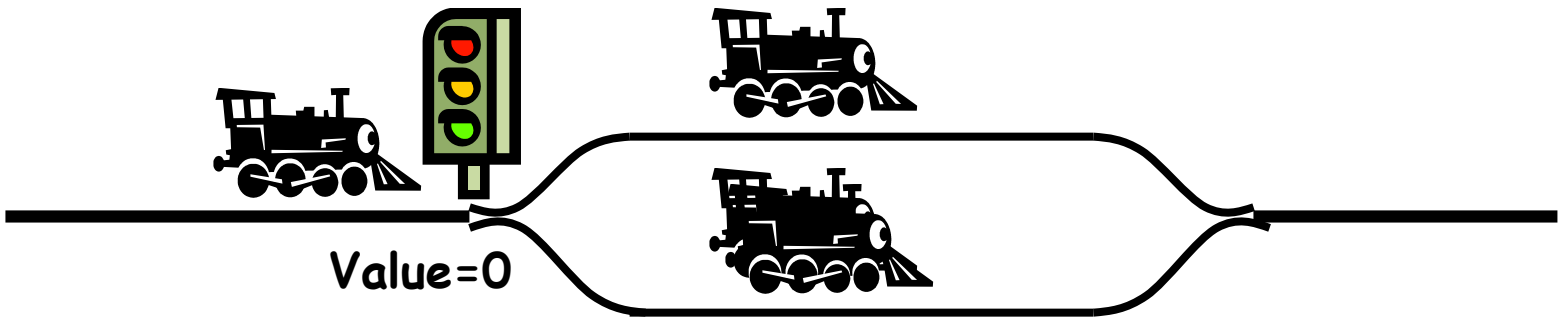
- Definition: a Semaphore has a non-negative integer value and supports the following two operations:
  - P(): an atomic operation that waits for semaphore to become positive, then decrements it by 1
    - Think of this as the wait() operation
  - V(): an atomic operation that increments the semaphore by 1, waking up a waiting P, if any
    - This of this as the signal() operation

- Semaphores were initially introduced to solve the producer – consumer paradigm

ltsirov@visteon.com

Visteon

- Semaphores are like integers, except
  - No negative values
  - Only operations allowed are P and V – can't read or write value, except to set it initially
  - Operations must be atomic
    - Two P's together can't decrement value below zero
    - Similarly, thread going to sleep in P won't miss wakeup from V – even if they both happen at same time
- Semaphore from railway analogy
  - Here is a semaphore initialized to 2 for resource control:



ltsirov@visteon.com

Visteon

Two uses of semaphores

- Mutual Exclusion (initial value = 1)
  - Also called “Binary Semaphore”.
  - Can be used for mutual exclusion:

```
semaphore.P(); // Lock the access
// Critical section goes here
semaphore.V(); // Unlock the access
```

- Scheduling Constraints (initial value = 0)
  - Locks are fine for mutual exclusion, but what if you want a thread to wait for something?
  - Example: suppose you had to implement ThreadJoin which must wait for thread to terminiate:

```
Initial value of semaphore = 0
ThreadJoin {
   semaphore.P();
}
ThreadFinish {
   semaphore.V();
}
```


ltsirov@visteon.com

Visteon

Instead of using semaphores for mutual exclusion we can use *mutexes.*

- Usage of a mutex:

  1. When a program is started, a mutex is created with a unique name

  2. After this stage, any thread(process) that needs the resource must lock the mutex from other threads while it is using the resource

  3. The mutex is set to unlock when the data is no longer needed or the routine is finished

ltsirov@visteon.com

Visteon

Instead of using semaphores for scheduling constraints we can use *condition variables (a.k.a*. CondVars).

- Condition Variable: a queue of threads waiting for something *inside* a critical section
  - Key idea: make it possible to go to sleep inside critical section by atomically releasing lock at time we go to sleep.
  - In other words, the lock is released, while one process is waiting on the CondVar, so another process can acquire the lock and signal the first process to continue its operation
  - Contrast to semaphores: Can’t wait inside critical section

ltsirov@visteon.com

Basic concepts of OS: Concurrency

Visteon

A synchronized queue example

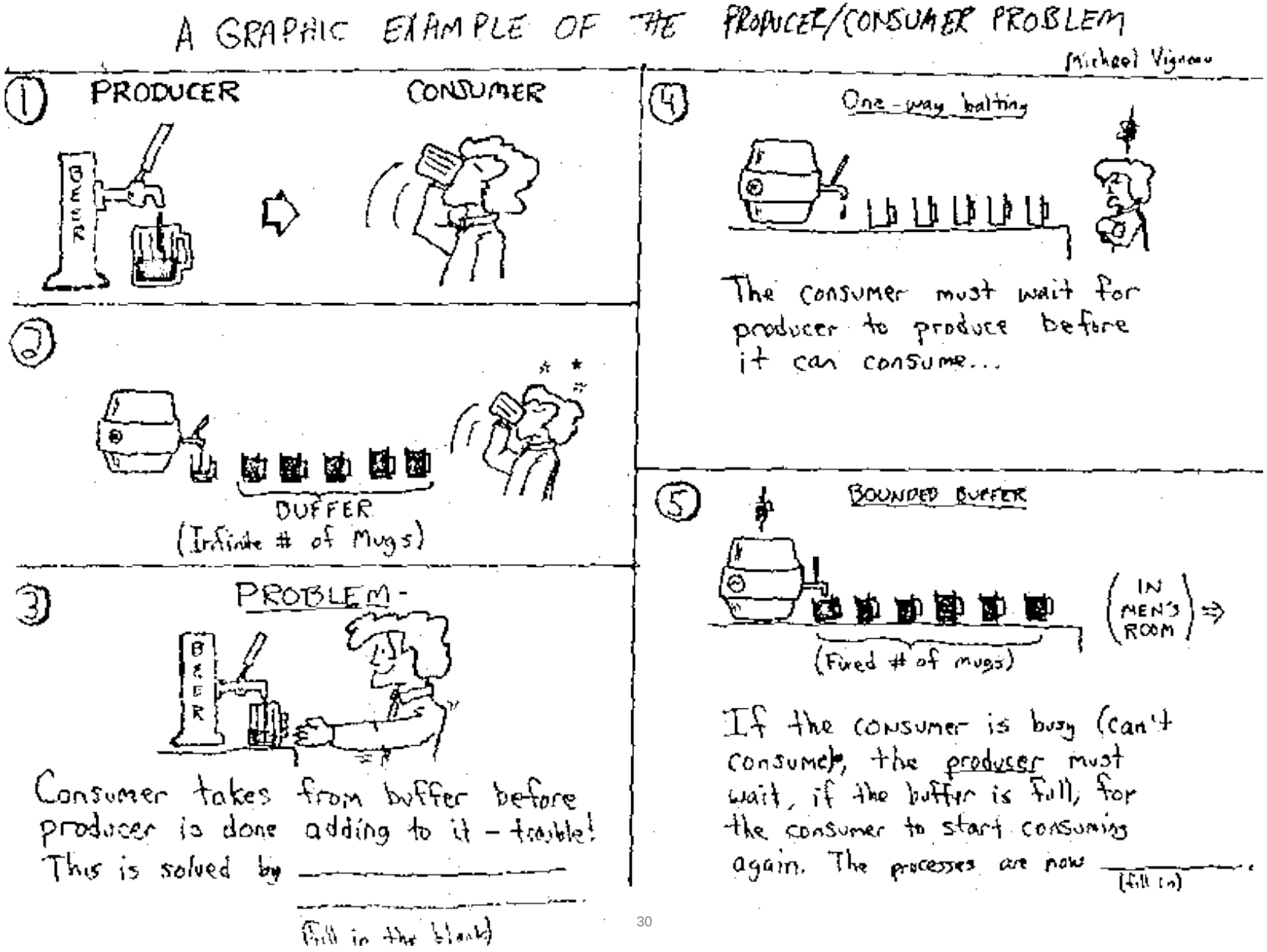
```
Mutex lock;
CondVar dataready;
Queue queue;


AddToQueue(item) {
   lock.Acquire();              // Get Lock
   queue.enqueue(item);         // Add item
   dataready.signal();          // Signal any
waiters
   lock.Release();              // Release Lock
}

RemoveFromQueue() {
   lock.Acquire();              // Get Lock
   while (queue.isEmpty()) {
      dataready.wait(&lock); // If nothing,
sleep
   }
   item = queue.dequeue();   // Get next item
   lock.Release();              // Release Lock
   return(item);
}
```


ltsirov@visteon.com

Basic concepts of OS: Concurrency example



ltsirov@visteon.com

Contents

Visteon

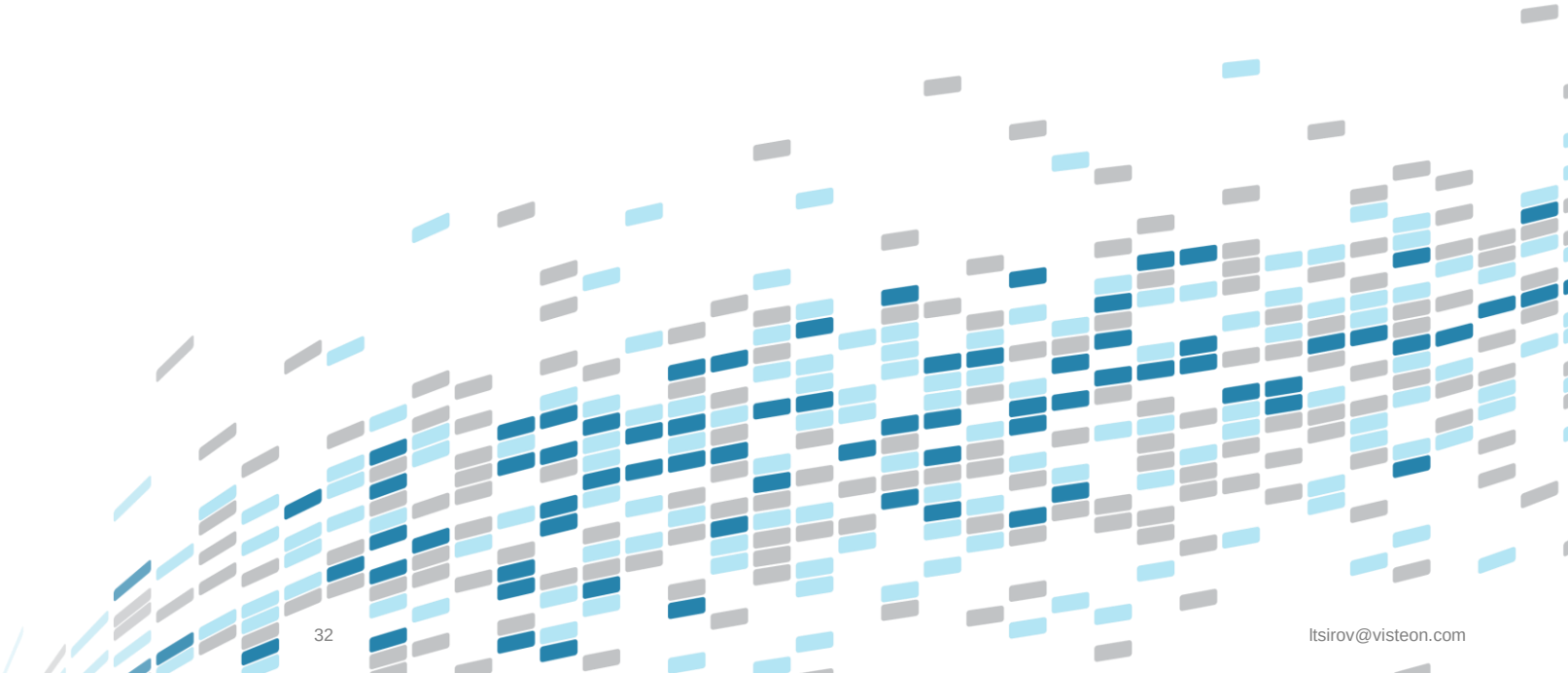
1. What is an operating system?
2. Kernel
3. System call
4. Concurrency – issues and solutions
5. **Multi process programming**
6. Multi thread programming
7. Advanced OS concepts
   - Scheduling(time sharing/time slicing)
   - Memory management & Virtual memory(address space)
   - I/O access
   - Interrupts
   - Real time operations
8. Inter process communication in UNIX

ltsirov@visteon.com

Multi process programming

Visteon

- What Is A Process
- Process Creation
  - The fork() System Call
- Child Process Termination
  - The wait() System Call
- Process Termination



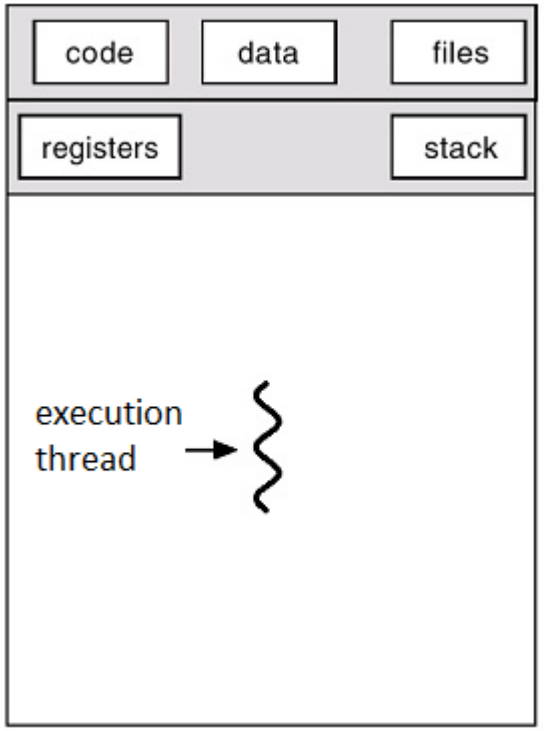
ltsirov@visteon.com

Visteon

What Is A Process?

UNIX definition:

- *An entity that executes a given piece of code, has its own execution stack, its own set of memory pages, its own file descriptors table, and a unique process ID.*



ltsirov@visteon.com

Visteon

What Is A Process?

A process is not a program! Several processes may be executing the same computer program at the same time, for the same user or for several different users.

ltsirov@visteon.com

Visteon

What Is A Process?

- It might be that many different processes will try to execute the same piece of code at the same time, perhaps trying to utilize the same resources, and we should be ready to accommodate such situations. This leads us to the concept of 'Re-entrancy'.

- Re-entrancy
  - The ability to have the same function (or part of a code) being in some phase of execution, more than once at the same time.

- This re-entrancy might mean that two or more processes try to execute this piece of code at the same time.

ltsirov@visteon.com

Visteon

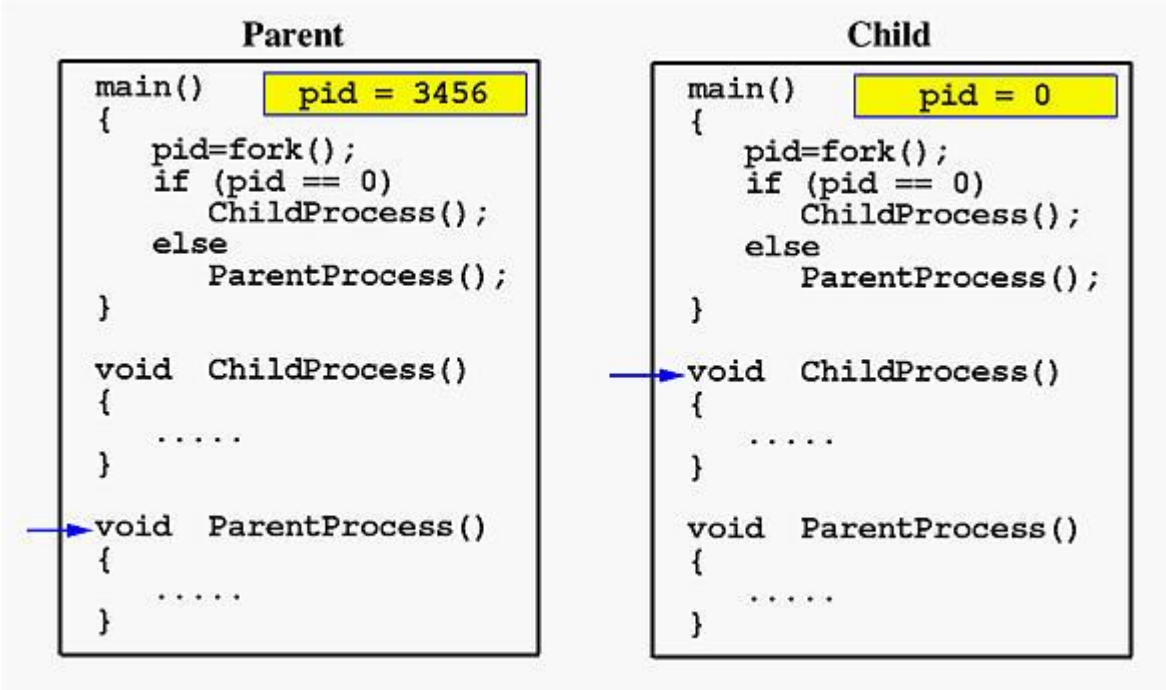
Process Creation

- The fork() system call is the basic way to create a new process. fork() is used to produce child shell.
- Returns twice(!!!!)
- fork() causes the current process to be split into two processes
  - a parent process
  - a child process.
- All of the memory pages used by the original process get duplicated during the fork() call, so both parent and child process see the exact same memory image.

ltsirov@visteon.com

Visteon

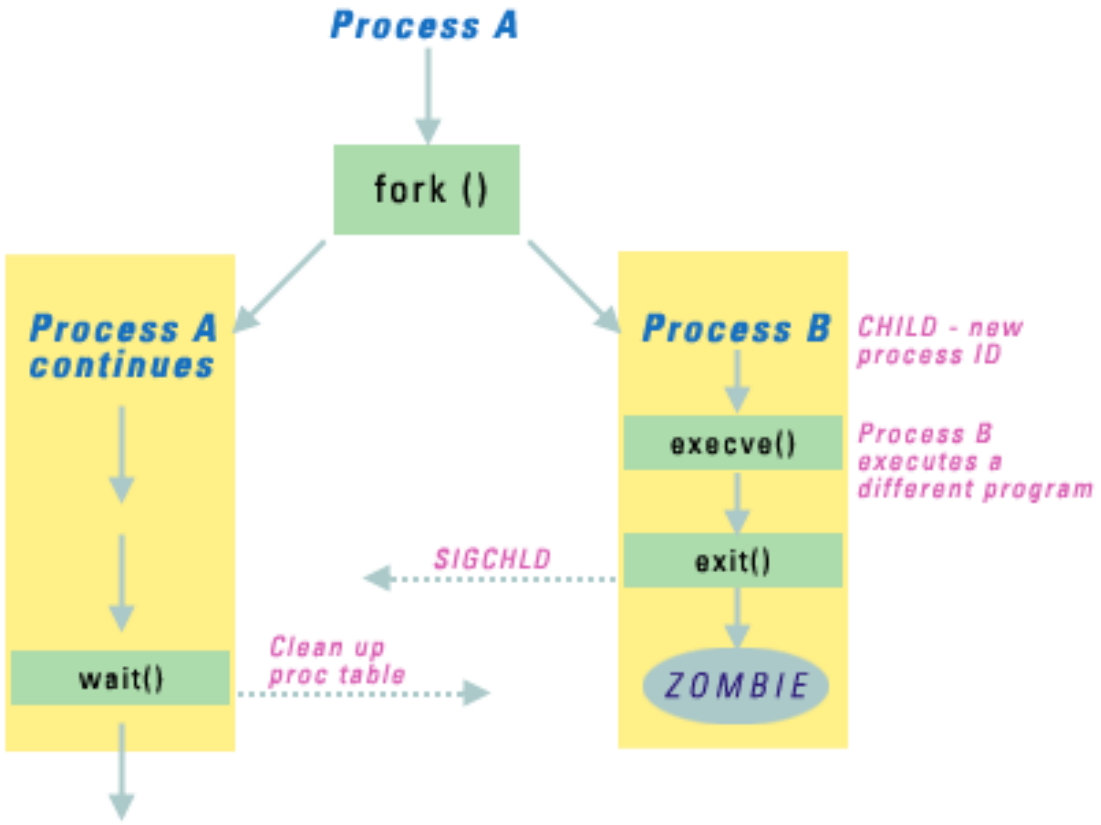
Process Creation



ltsirov@visteon.com

Visteon

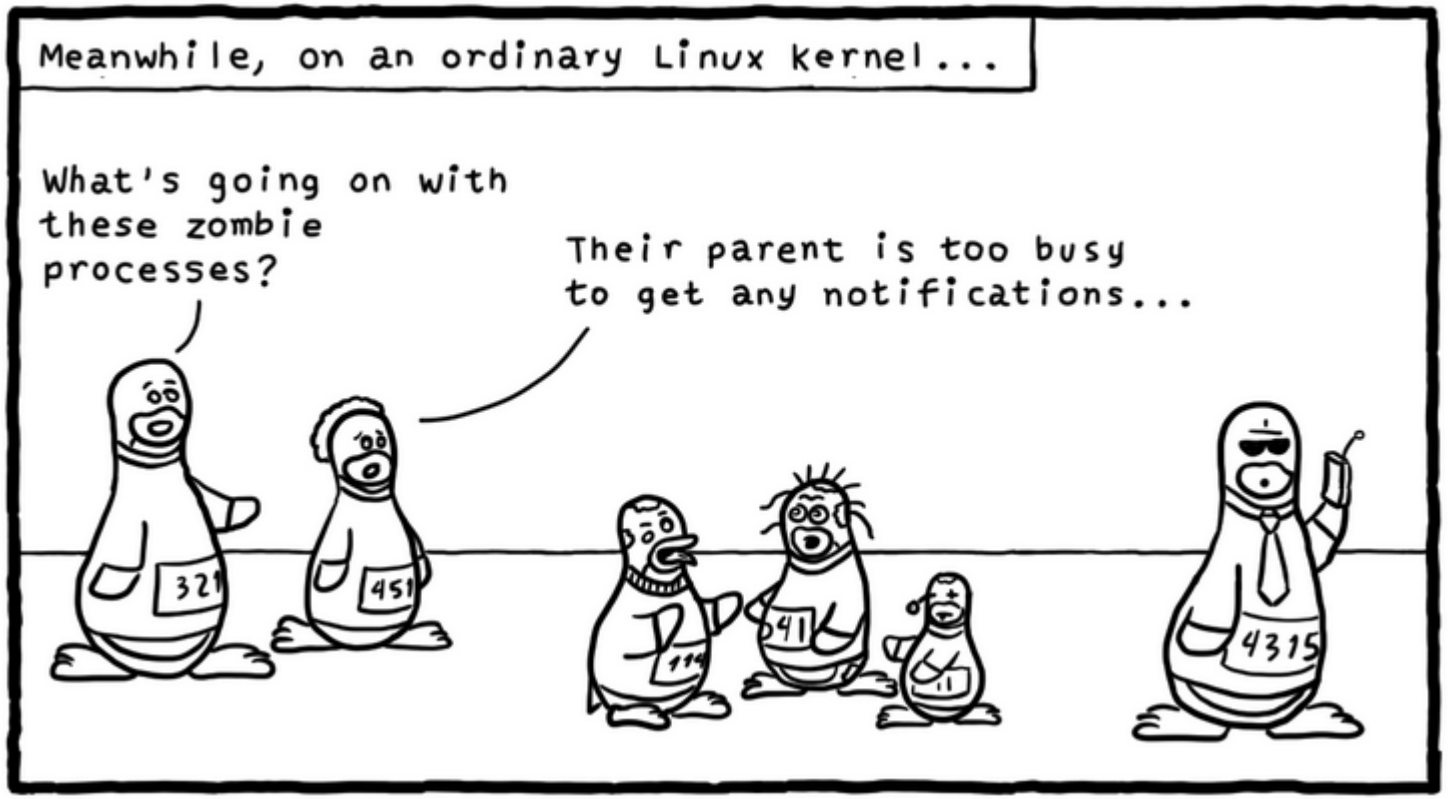
Process Creation



ltsirov@visteon.com

Visteon

Child Process Termination



ltsirov@visteon.com

Visteon

Child Process Termination

Once we have created a child process, there are two possibilities.

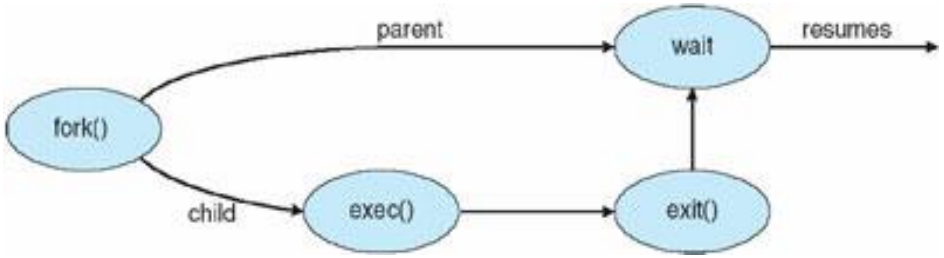
- When a child process exits, it is not immediately cleared off the process table. Instead, a signal is sent to its parent process, which needs to acknowledge it's child's death, and only then the child process is completely removed from the system. In the duration before the parent's acknowledgment and after the child's exit, the child process is in a state called "**zombie**".
- When a process exits (terminates), if it had any child processes, they become orphans. An orphan process is automatically inherited by the 'init' process (process number 1 on normal Unix systems), and becomes a child of this 'init' process. This is done to ensure that when the process terminates, it does not turn into a zombie, because 'init' is written to properly acknowledge the death of its child processes.
- When the parent process is not properly coded, the child remains in the zombie state forever.

ltsirov@visteon.com

Visteon

The wait() System Call

- The simple way of a process to acknowledge the death of a child process is by using the wait() system call.
- When wait() is called, the process is suspended until one of its child processes exits, and then the call returns with the exit status of the child process.
- If it has a zombie child process, the call returns immediately, with the exit status of that process.



ltsirov@visteon.com

Visteon

Process termination

int **kill** *(pid_t pid, int signum)*

The kill function can be used to send a signal to another process. In spite of its name, it can be used for a lot of things other than causing a process to terminate. Some examples of situations where you might want to send signals between processes are:

- A parent process starts a child to perform a task—perhaps having the child running an infinite loop—and then terminates the child when the task is no longer needed.
- A process executes as part of a group, and needs to terminate or notify the other processes in the group when an error or other event occurs.
- Two processes need to synchronize while working together.

ltsirov@visteon.com

Visteon

Process termination

Macro: *int* **SIGTERM**

- The SIGTERM signal is a generic signal used to cause program termination. Unlike SIGKILL, this signal can be blocked, handled, and ignored. It is the normal way to politely ask a program to terminate.
- The shell command kill generates SIGTERM by default.

Macro: *int* **SIGINT**

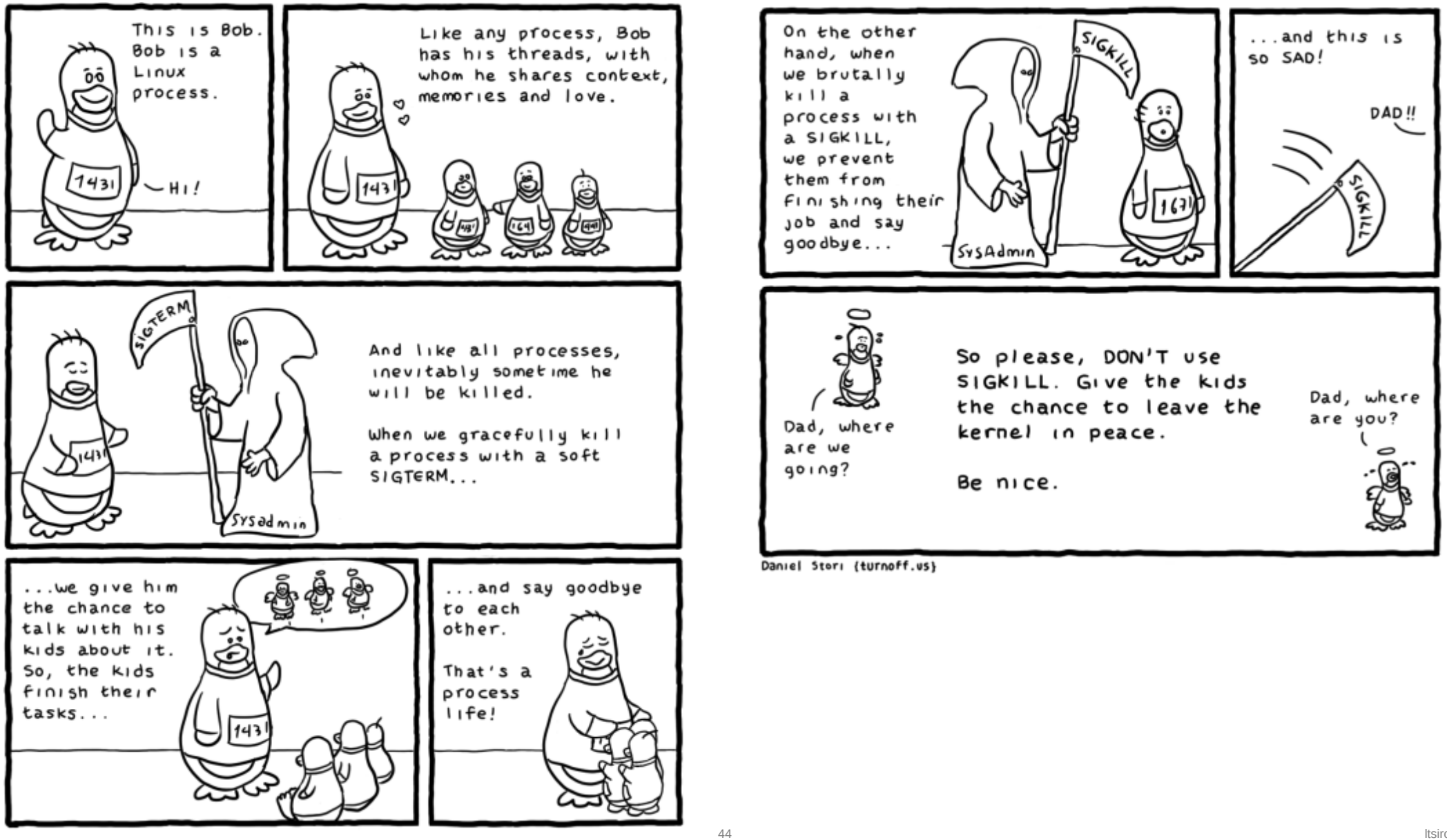
- The SIGINT (“program interrupt”) signal is sent when the user types the INTR character (normally C-c)

Macro: *int* **SIGKILL**

- The SIGKILL signal is used to cause immediate program termination. It cannot be handled or ignored, and is therefore always fatal. It is also not possible to block this signal.
- This signal is usually generated only by explicit request. Since it cannot be handled, you should generate it only as a last resort, after first trying a less drastic method such as C-c or SIGTERM. If a process does not respond to any other termination signals, sending it a SIGKILL signal will almost always cause it to go away.
- In fact, if SIGKILL fails to terminate a process, that by itself constitutes an operating system bug which you should report.

ltsirov@visteon.com

Multi process programming



Daniel Stori {turnoff.us}

ltsirov@visteon.com

Contents

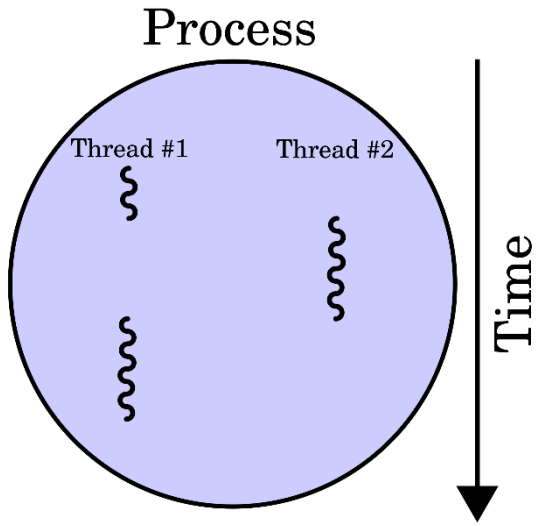
1. What is an operating system?
2. Kernel
3. System call
4. Concurrency – issues and solutions
5. Multi process programming
6. **Multi thread programming**
7. Advanced OS concepts
   - Scheduling(time sharing/time slicing)
   - Memory management & Virtual memory(address space)
   - I/O access
   - Interrupts
   - Real time operations
8. Inter process communication in UNIX

ltsirov@visteon.com

Multi thread programming

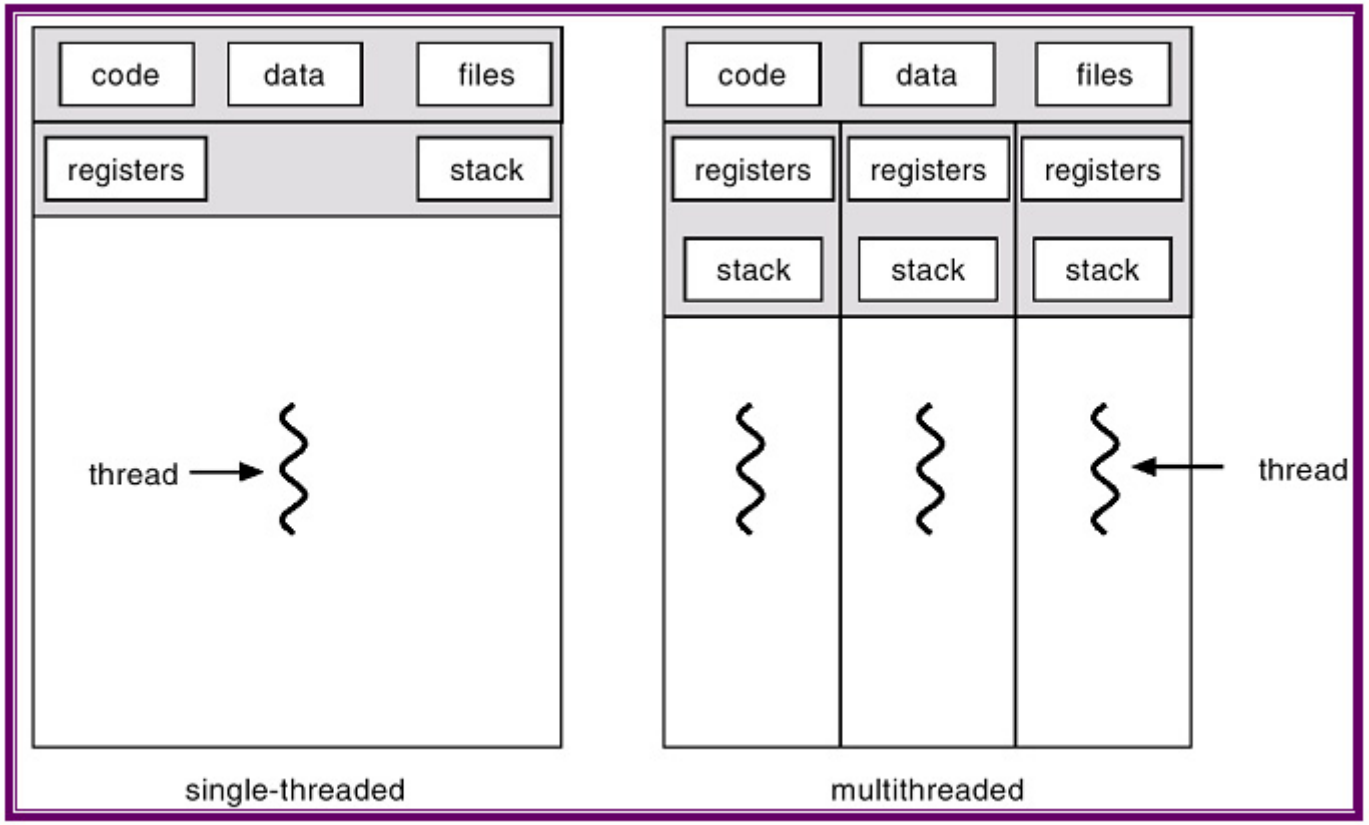
Visteon

- What is a thread?
- *Definition:*
  - *a thread is defined as an independent stream of instructions that can be scheduled to run as such by the operating system.*
- Processes vs. threads



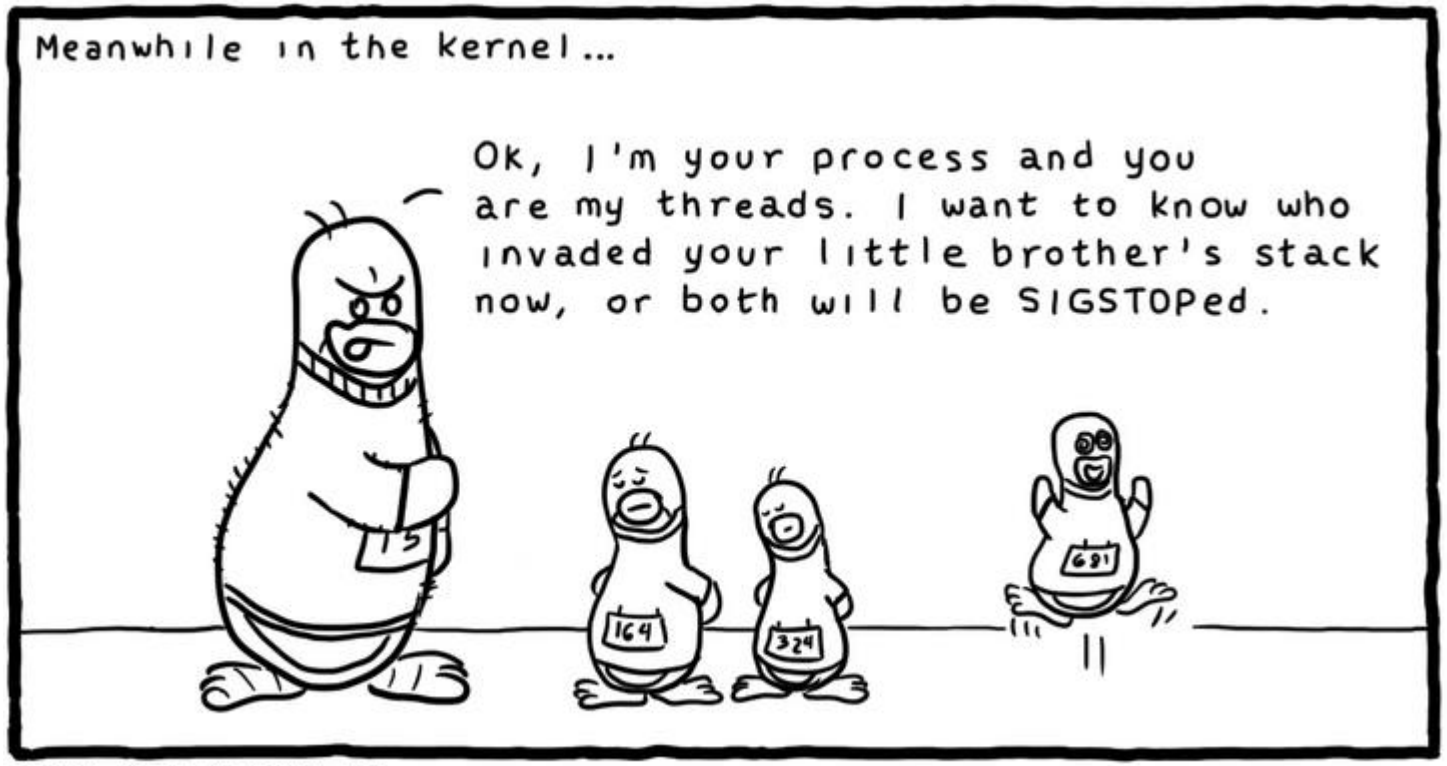
ltsirov@visteon.com

Multi thread programming



ltsirov@visteon.com

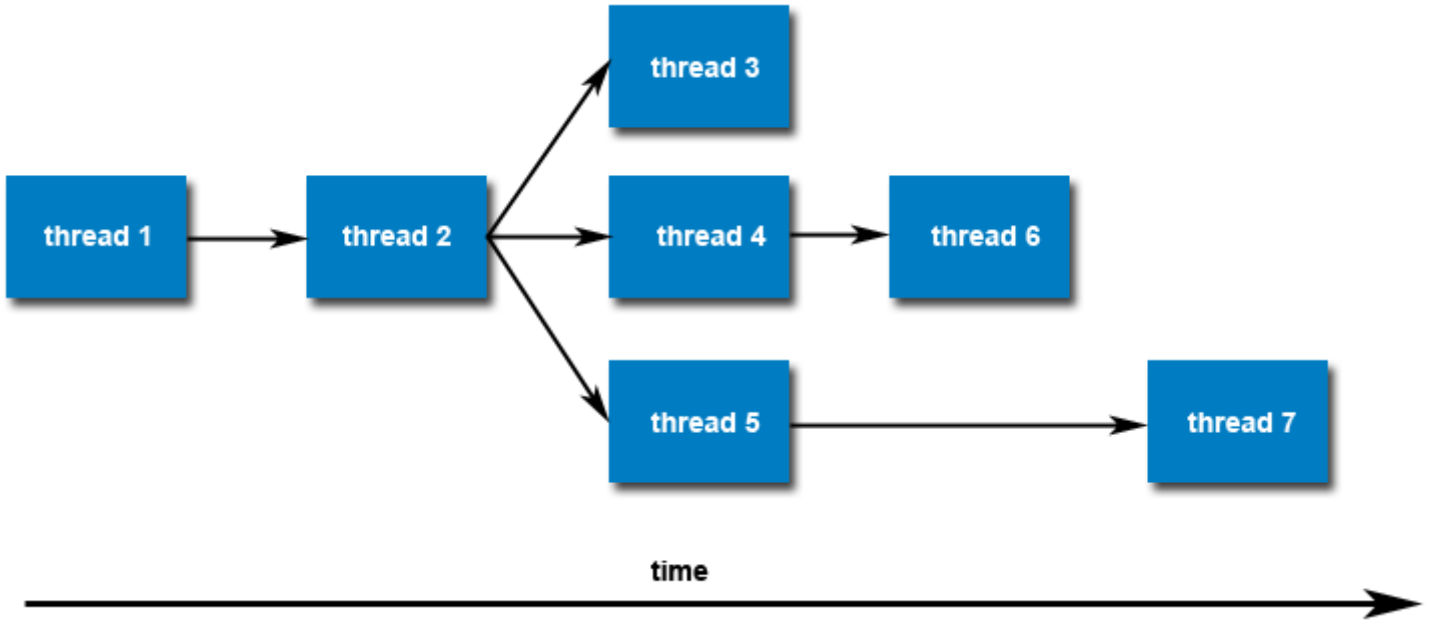
Multi thread programming



ltsirov@visteon.com

Multi thread programming

Visteon



ltsirov@visteon.com

Multi thread programming

Visteon

- POSIX threads
  - The pthread library is encapsulating all of the thread functionality in the POSIX based OS(e.g. Linux, QNX…)

- Thread manipulation
  - pthread_create (thread,attr,start_routine,arg)
  - pthread_cancel (thread)

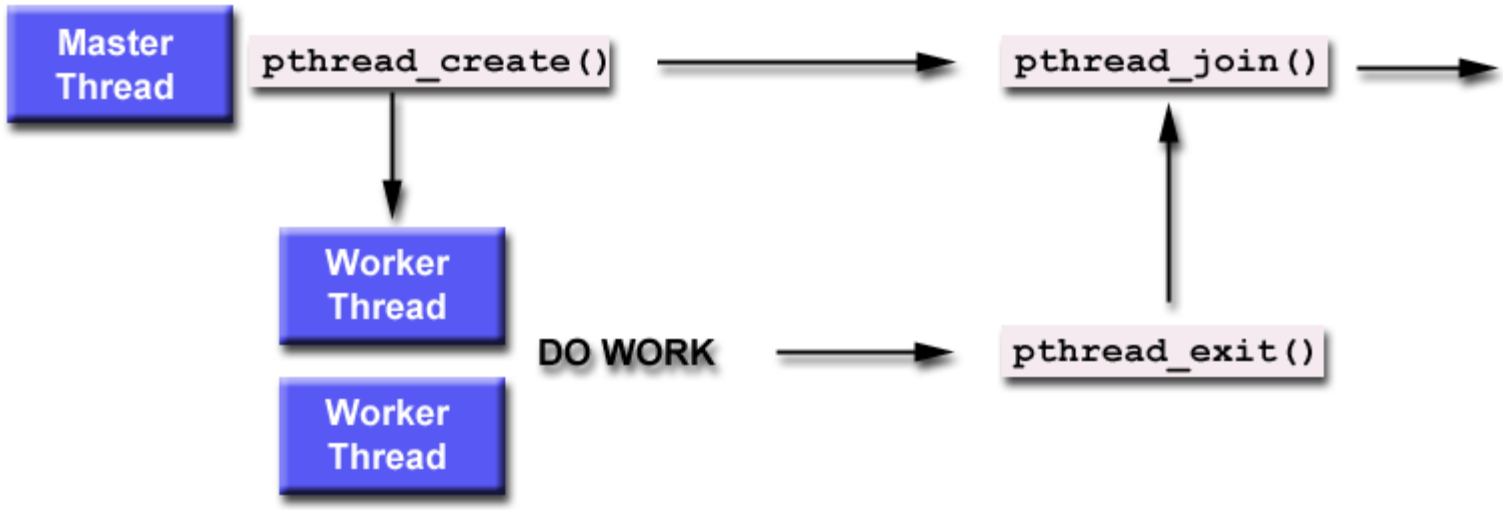
Keep in mind: All POSIX thread functions are part of the libpthread.so

ltsirov@visteon.com

Multi thread programming

Visteon

- Thread manipulation
  - pthread_exit (status)
  - pthread_join (threadid,status)



ltsirov@visteon.com

Multi thread programming

Visteon

3 main thread synchronization primitives in pthreads library:

- semaphore
- mutex
- condvar

- Mutex manipulation
  - pthread_mutex_init (mutex,attr)
  - pthread_mutex_destroy (mutex)
  - pthread_mutex_lock (mutex)
  - pthread_mutex_unlock (mutex)

ltsirov@visteon.com

Multi thread programming

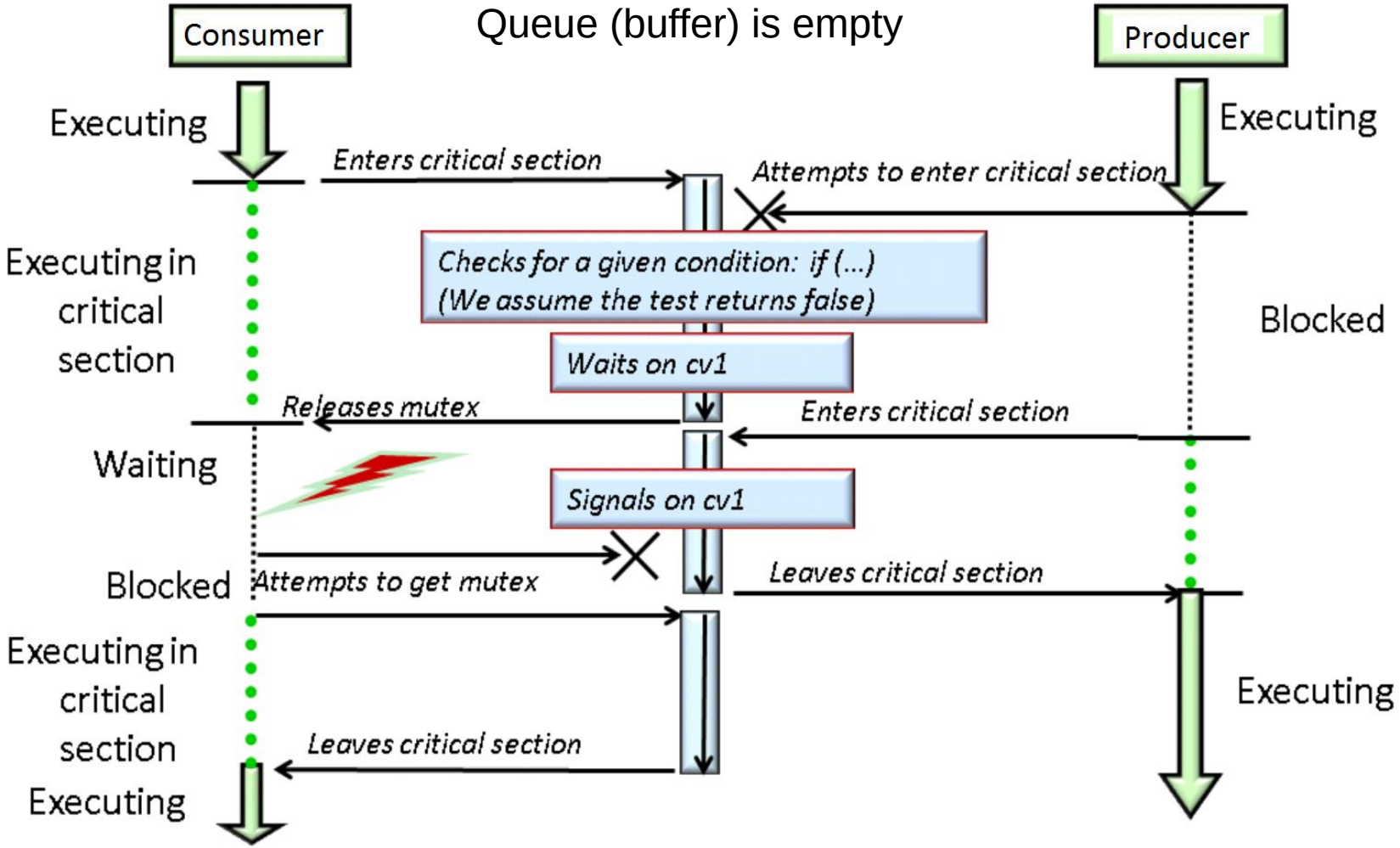
Visteon

- Condvar manipulation
  - pthread_cond_init (condition,attr)
  - pthread_cond_destroy (condition)
  - pthread_cond_wait (condition,mutex)
  - pthread_cond_signal (condition)
  - pthread_cond_broadcast (condition)

pthread_cond_wait() takes a mutex as an argument.

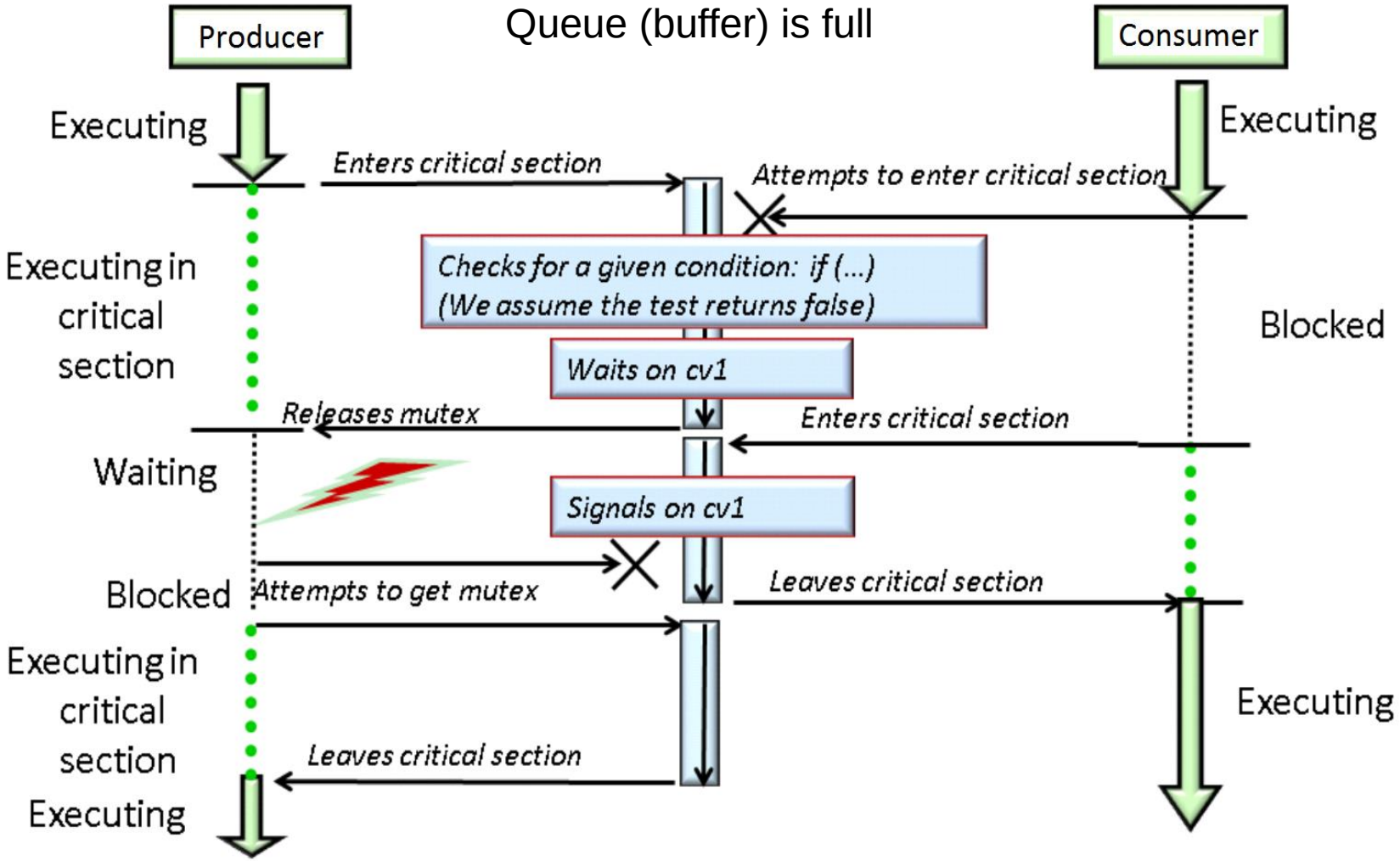
ltsirov@visteon.com

Visteon



ltsirov@visteon.com

Visteon

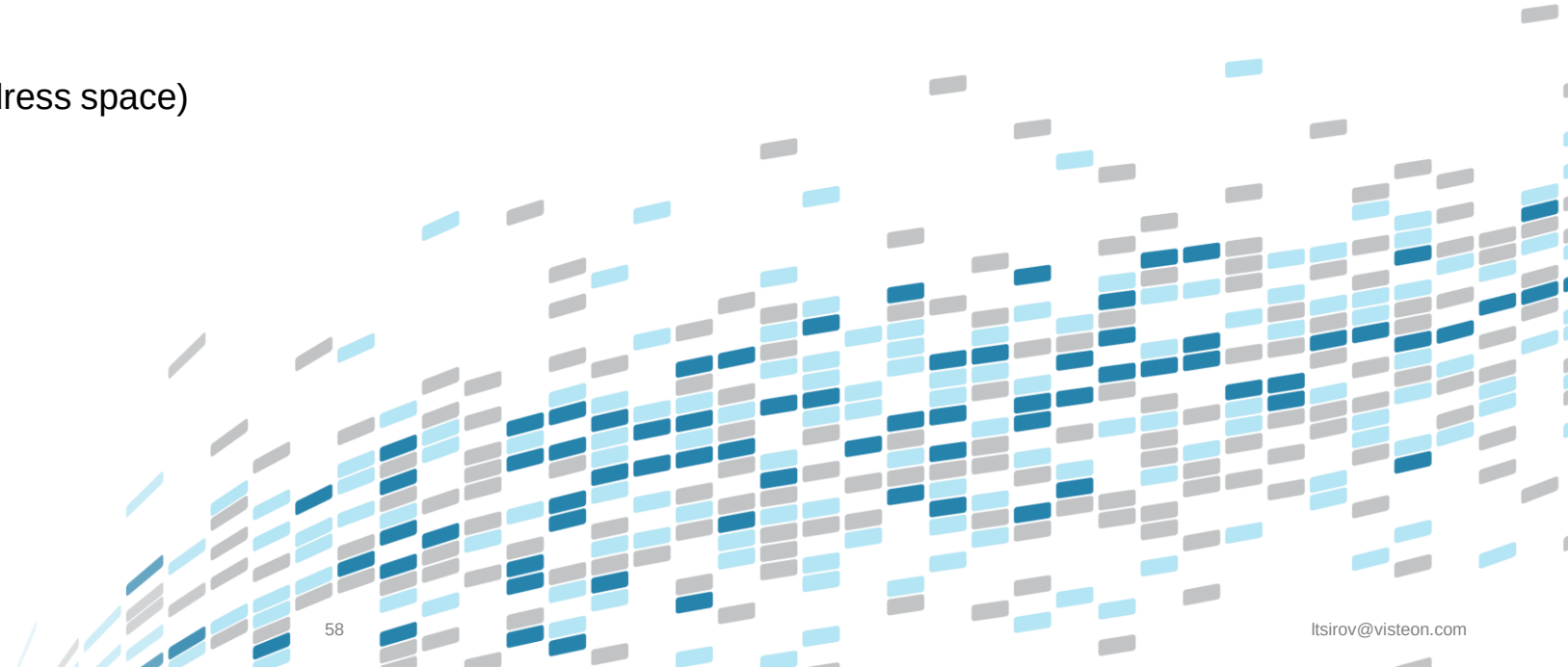


ltsirov@visteon.com

Contents

Visteon

1. What is an operating system?
2. Kernel
3. System call
4. Concurrency – issues and solutions
5. Multi process programming
6. Multi thread programming
7. **Concepts of OS**
   - Scheduling(time sharing/time slicing)
   - Memory management & Virtual memory(address space)
   - I/O access
   - Interrupts
   - Real time operations
8. Inter process communication in UNIX

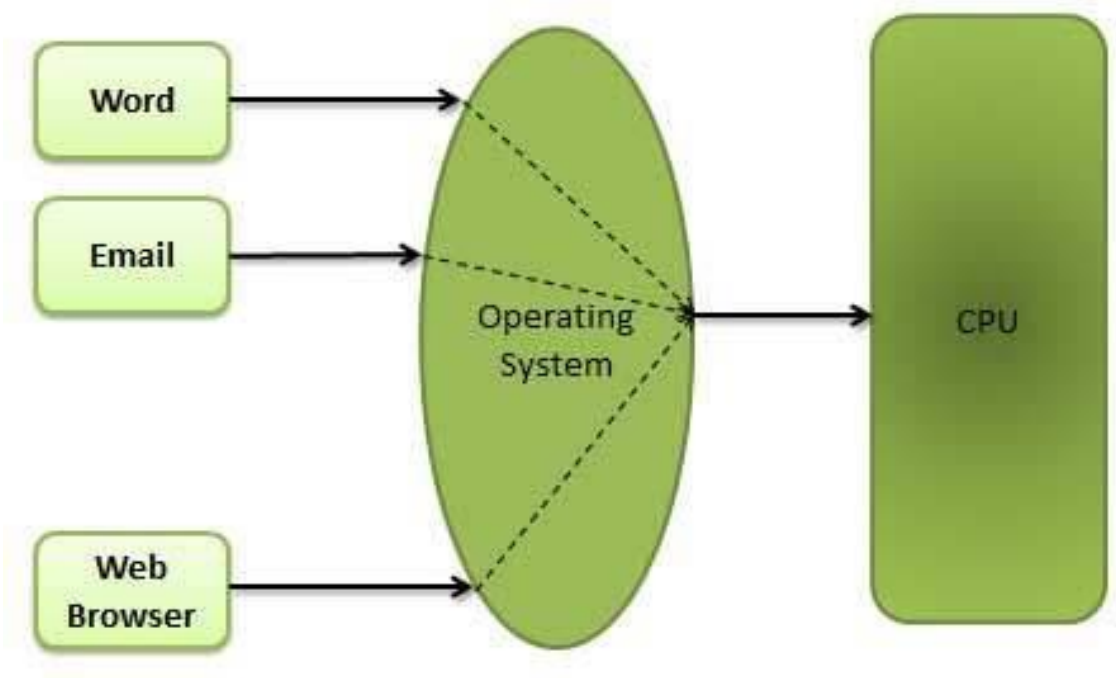


ltsirov@visteon.com

Concepts of OS: Scheduling

Visteon

A operating system can be **multitasking** or **multiprogramming** - appear to be running several processes at once even on one physical CPU. This can be achieved by giving each process a slice of time on a processor and a slice of memory for running and allocating resources as necessary.

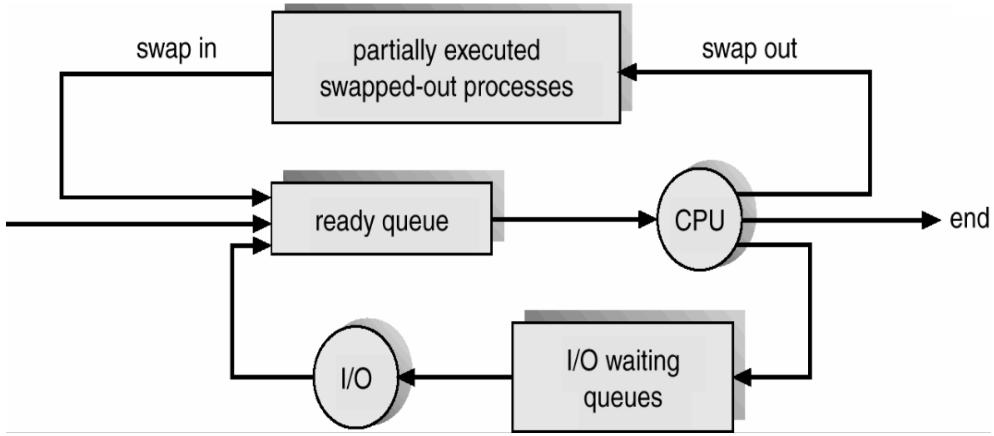


ltsirov@visteon.com

Concepts of OS: Scheduling

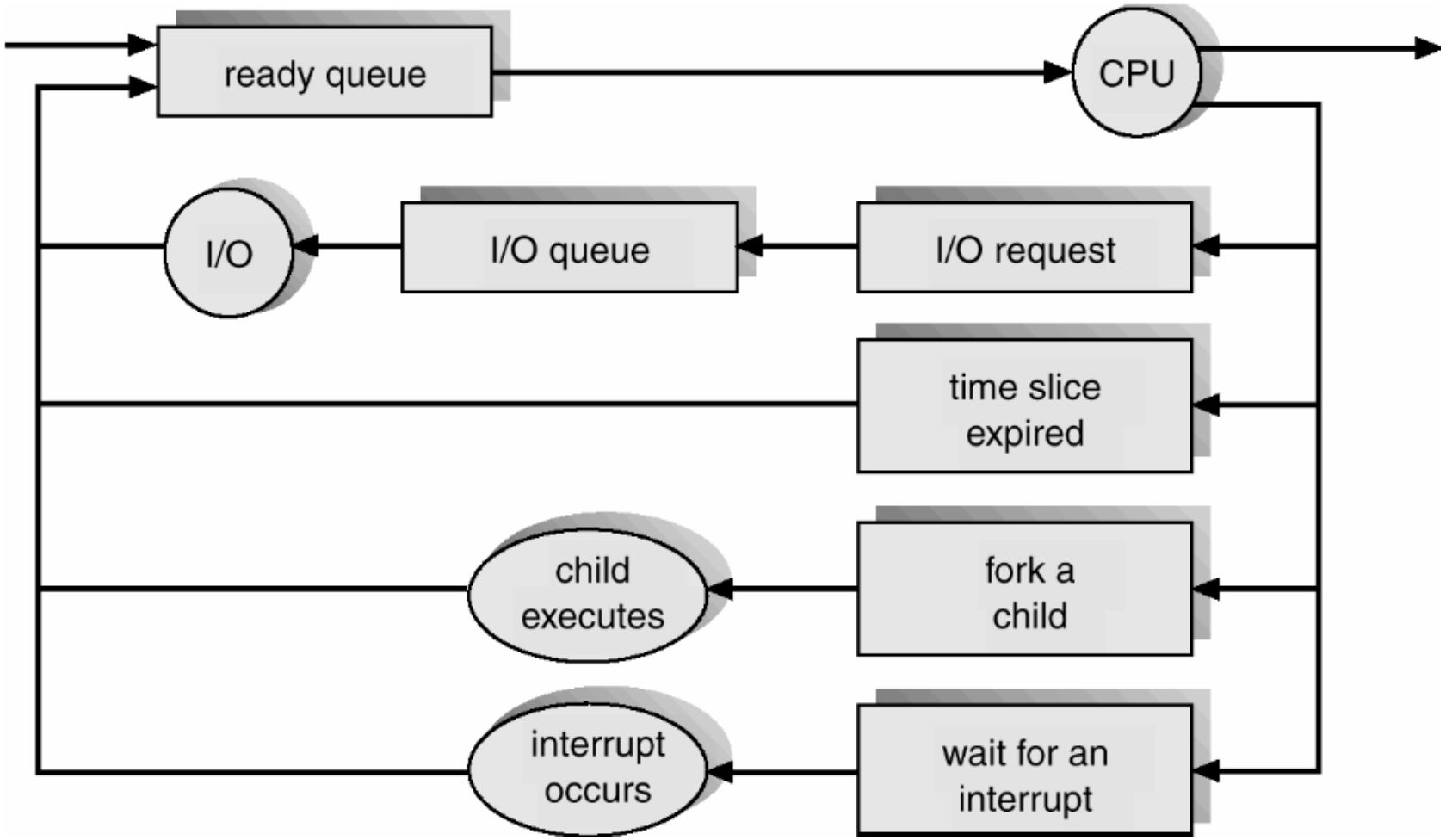
A **scheduler** allocates time. It decides when a process has used up enough time and should be forced to free the processor and be swapped for another process.

- Often processes are forced off a processor before their allocated time is up because they are doing I/O and have to wait for I/O to complete - devices are typically very slow compared to the CPU.
- Whenever the CPU becomes idle, it is the job of the scheduler to select another process from the ready queue to run next.
- The storage structure for the ready queue and the algorithm used to select the next process are not necessarily a FIFO queue.



ltsirov@visteon.com

Concepts of OS: Scheduling



ltsirov@visteon.com

Visteon

Preemptive vs. non-preemptive scheduling

- Non-preemptive scheduling
  - Once in running state, process will continue
  - Potential to monopolize the CPU
  - May voluntarily yield the CPU

- Preemptive scheduling
  - Currently running process may be interrupted by OS and put into ready state
  - Timer interrupts required (for IRP)
  - Incurs context switches

ltsirov@visteon.com

Concepts of OS: Scheduling

Visteon

The **dispatcher** is the module that gives control of the CPU to the process selected by the scheduler. This function involves:

- Switching context.
- Switching to user mode.
- Jumping to the proper location in the newly loaded program.
- The dispatcher needs to be as fast as possible, as it is run on every context switch.

The time consumed by the dispatcher is known as dispatch latency.

Process scheduling is an essential part of a Multiprogramming operating system. Such operating systems allow more than one process to be loaded into the executable memory at a time and loaded process shares the CPU using time multiplexing.

The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy, called scheduling algorithm.

ltsirov@visteon.com

Concepts of OS: Scheduling

Visteon

The most common scheduling algorithms are:

- **First-Come First-Serve**
- **Shortest-Job-First**
- **Round Robin**
- **Priority Scheduling**

There are several different criteria to consider when trying to select scheduling algorithm for a particular situation and environment:

- **CPU utilization** - Ideally the CPU would be busy 100% of the time, so as to waste 0 CPU cycles. On a real system CPU usage should range from 40% (lightly loaded) to 90% (heavily loaded.)
- **Throughput** - Number of processes completed per unit time. May range from 10 / second to 1 / hour depending on the specific processes.
- **Turnaround time** - Time required for a particular process to complete, from submission time to completion. (Wall clock time.)
- **Waiting time** - How much time processes spend in the ready queue waiting their turn to get on the CPU.
  - **Average load** - The average number of processes sitting in the ready queue waiting their turn to get into the CPU. Reported in 1-minute, 5-minute, and 15-minute averages by "uptime" and "who".
- **Response time** - The time taken in an interactive program from the issuance of a command to the commence of a response to that command.

ltsirov@visteon.com

Visteon

First-Come First-Serve

- FCFS is very simple - Just a FIFO queue, like customers waiting in line at the bank or the post office or at a copying machine.
- Unfortunately, however, FCFS can yield some very long average wait times, particularly if the first process to get there takes a long time.

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

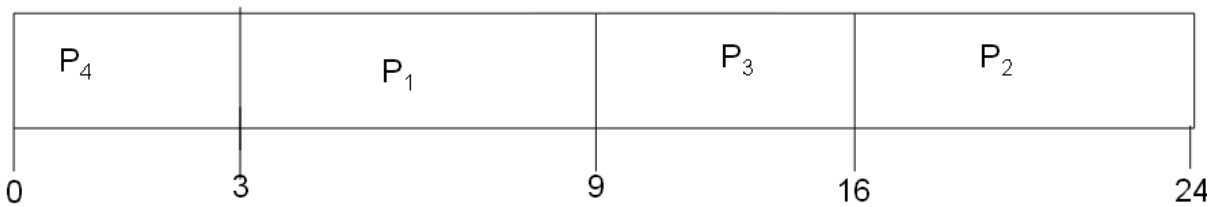
ltsirov@visteon.com

Visteon

Shortest-Job-First

- The idea behind the SJF algorithm is to pick the quickest fastest little job that needs to be done, get it out of the way first, and then pick the next smallest fastest job to do next.
- SJF is optimal - gives minimum average waiting time for a given set of processes

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3



- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

- The difficulty is knowing the length of the next job(CPU request)

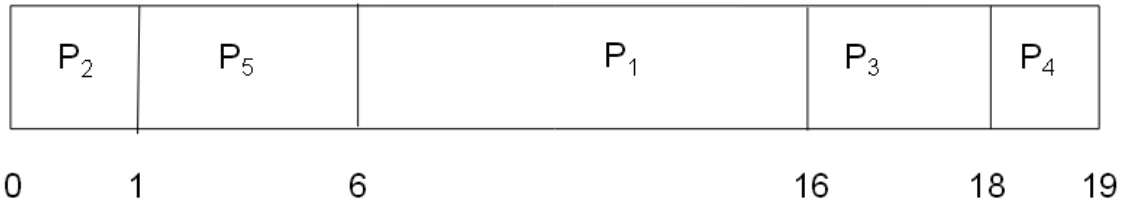
ltsirov@visteon.com

Visteon

Priority Scheduling

- Priority scheduling is a more general case of SJF, in which each job is assigned a priority and the job with the highest priority gets scheduled first.

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2



- Average waiting time = 8.2 msec

- **Problem**: Starvation - low priority processes may never execute
- **Solution**: Aging - as time progresses, increase the priority of the process

ltsirov@visteon.com

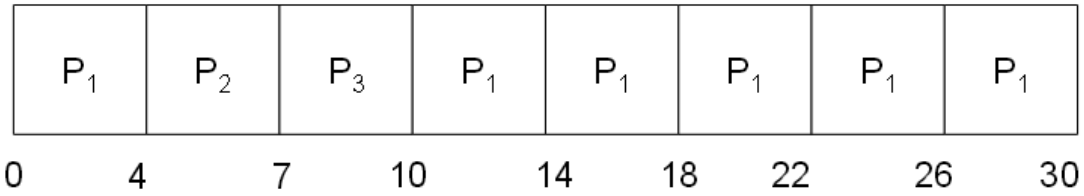
Visteon

Round Robin

- RR is similar to FCFS scheduling, except that CPU bursts are assigned with limits called ***time quantum*.**
- When a process is given the CPU, a timer is set for whatever value has been set for a time quantum.
  - If the process finishes its burst before the time quantum timer expires, then it is swapped out of the CPU just like the normal FCFS algorithm.
  - If the timer goes off first, then the process is swapped out of the CPU and moved to the back end of the ready queue.
- The ready queue is maintained as a circular queue, so when all processes have had a turn, then the scheduler gives the first process another turn, and so on.

Process	Burst Time
P_1	24
P_2	3
P_3	3

q = 4



- Typically, higher average turnaround than SJF, but better *response*
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 usec

ltsirov@visteon.com

Contents

Visteon

1. What is an operating system?
2. Kernel
3. System call
4. Concurrency – issues and solutions
5. Multi process programming
6. Multi thread programming
7. Concepts of OS
   - Scheduling(time sharing/time slicing)
   - **Memory management & Virtual memory(address space)**
   - I/O access
   - Interrupts
   - Real time operations
8. Inter process communication in UNIX

ltsirov@visteon.com

Concepts of OS: Memory management

Visteon

User's program must be brought into memory and placed within a process for it to be run.

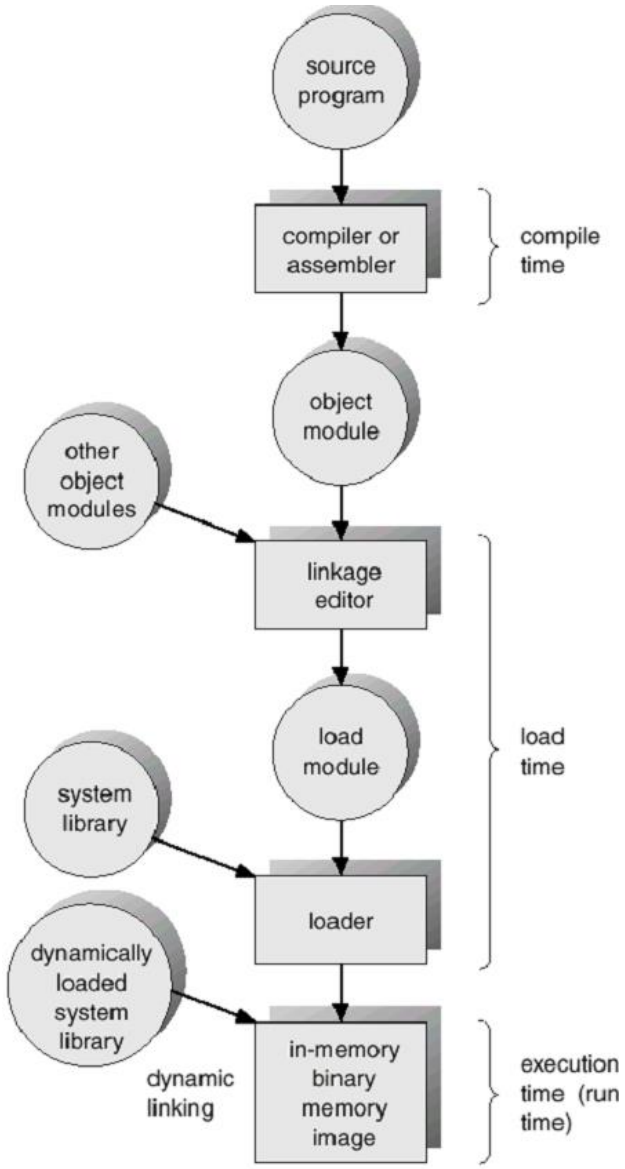
- *Input queue* – collection of processes on the disk that are waiting to be brought into memory to run the program.
- User programs go through several steps before being run

Address binding of instructions and data to memory addresses can happen at three different stages.

- Compile time: If memory location is known ,absolute code can be generated; must recompile code if starting location changes.
- Load time: Must generate relocatable code if memory location is not known at compile time.
- Execution time: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., base and limit registers).

ltsirov@visteon.com

Concepts of OS: Memory management



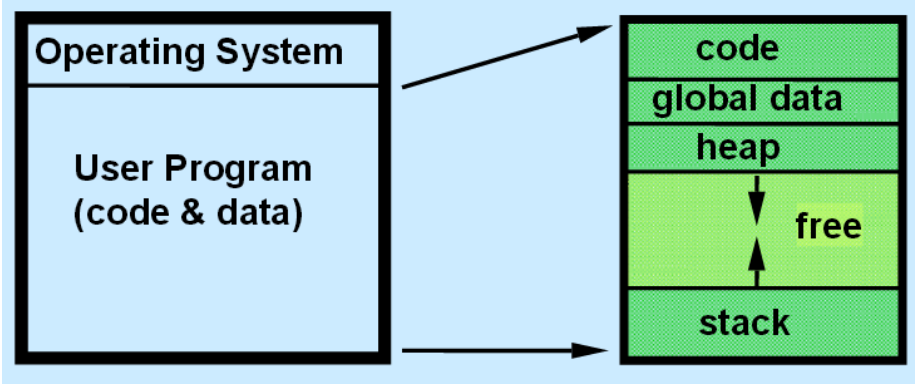
ltsirov@visteon.com

Concepts of OS: Memory management

Visteon

- The concept of a logical *address space* that is bound to a separate *physical address space* is central for proper memory management.
  - *Logical address* – generated by the CPU; also referred to as *virtual address*.
  - *Physical address* – address seen by the memory unit.

- Logical and physical addresses are the same in compile time and load-time address-binding schemes;
- Logical (virtual) and physical addresses differ in execution-time address-binding scheme.

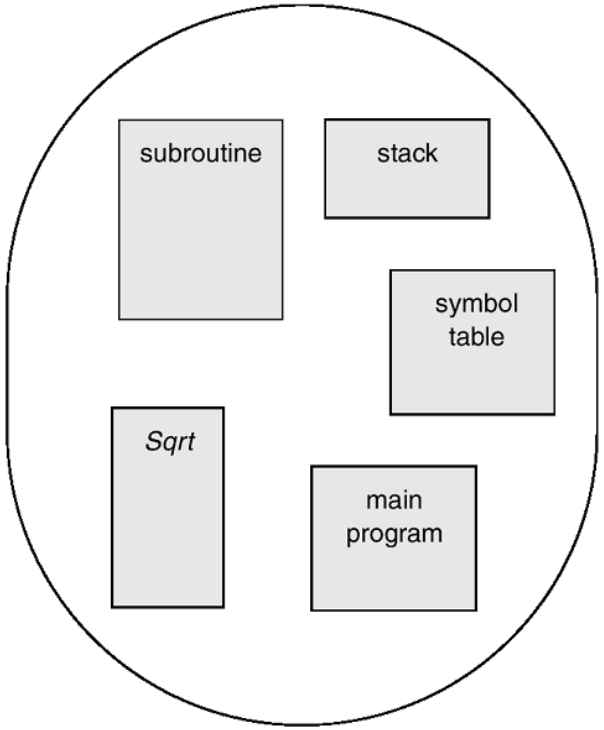


ltsirov@visteon.com

Concepts of OS: Memory management

User's program is a collection of segments in the logical address space. A segment is a logical unit such as:

- main program,
- procedures,
- functions,
- methods,
- objects,
- local variables,
- global variables,
- common block,
- stack,
- symbol table,
- arrays

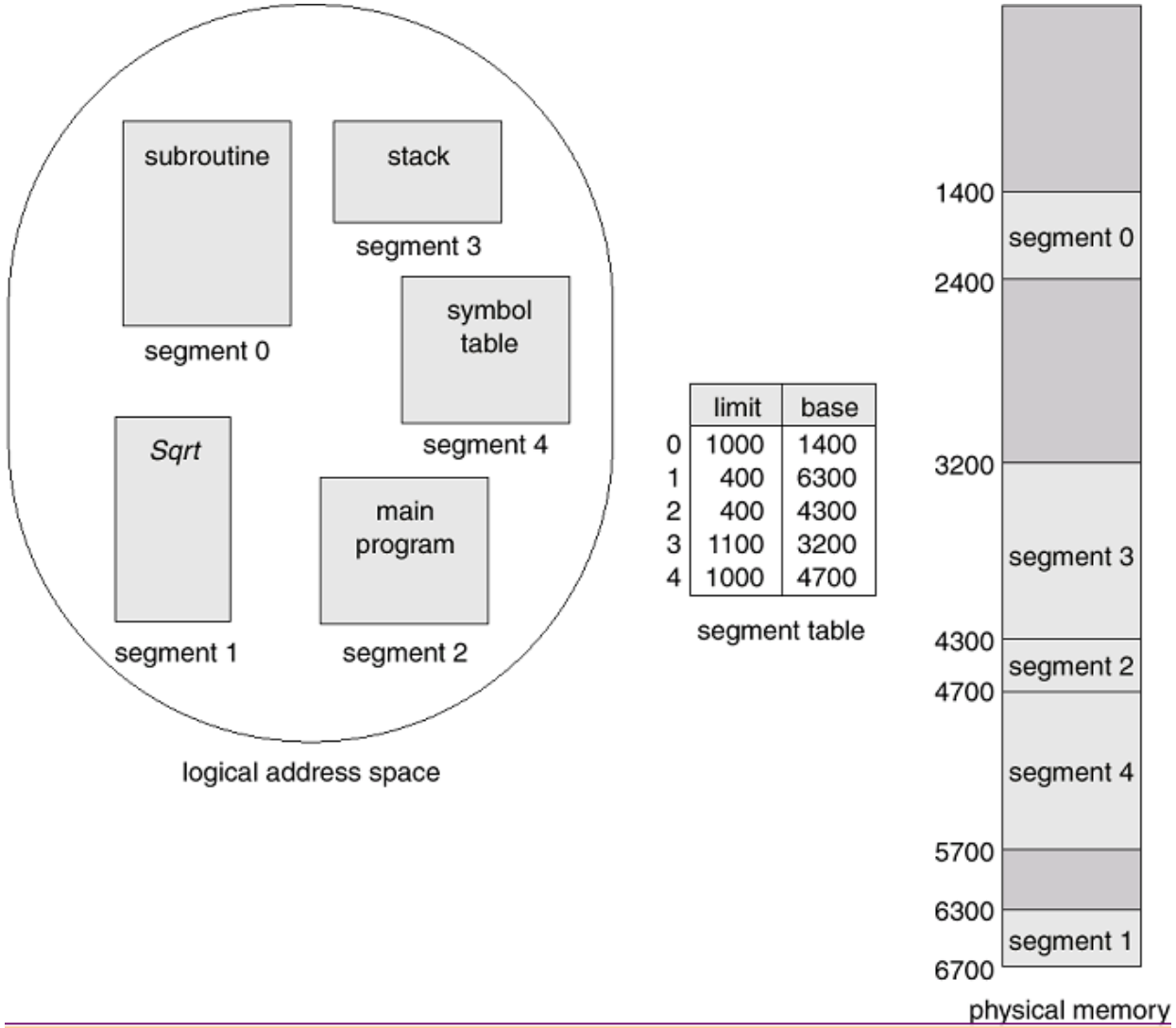


logical address space

ltsirov@visteon.com

Concepts of OS: Memory management

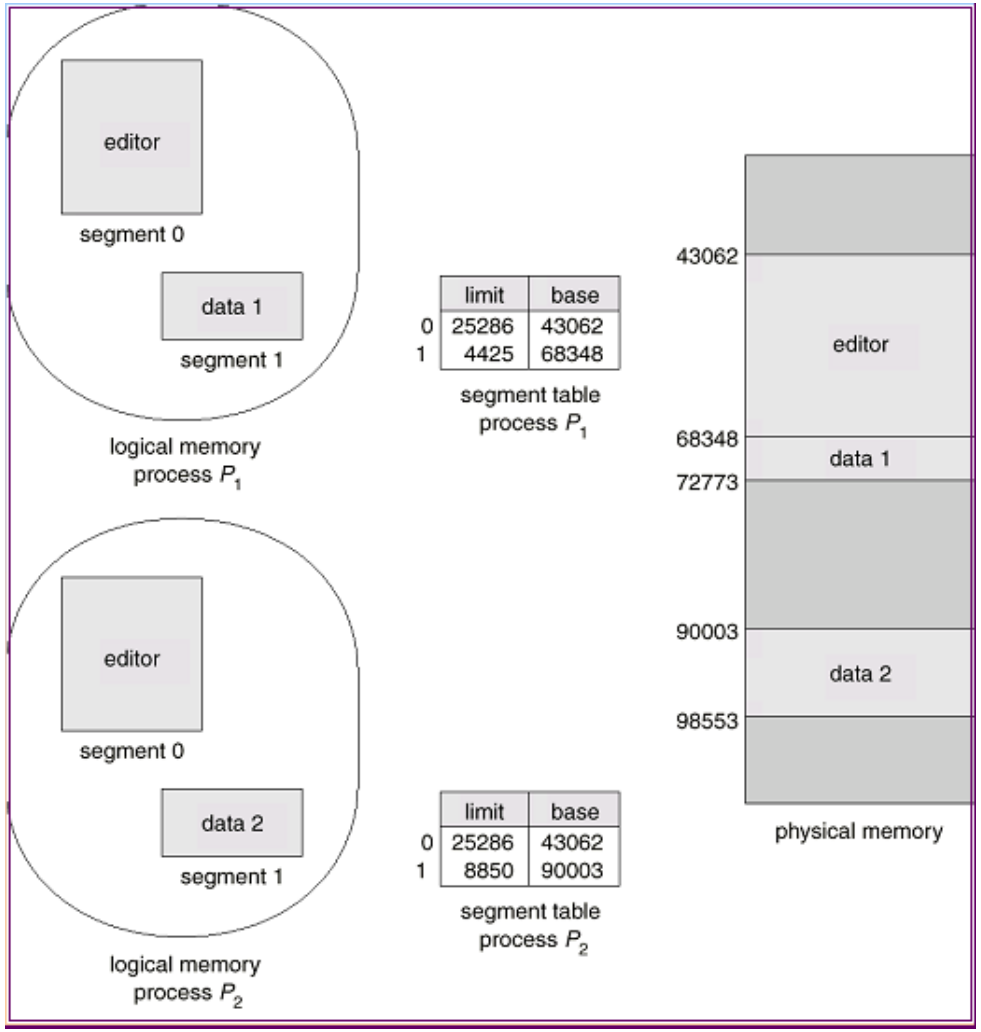
- Logical address space ≠ physical address



ltsirov@visteon.com

Concepts of OS: Memory management

Some program segments could be shared between several processes
(e.g. Shared libraries)



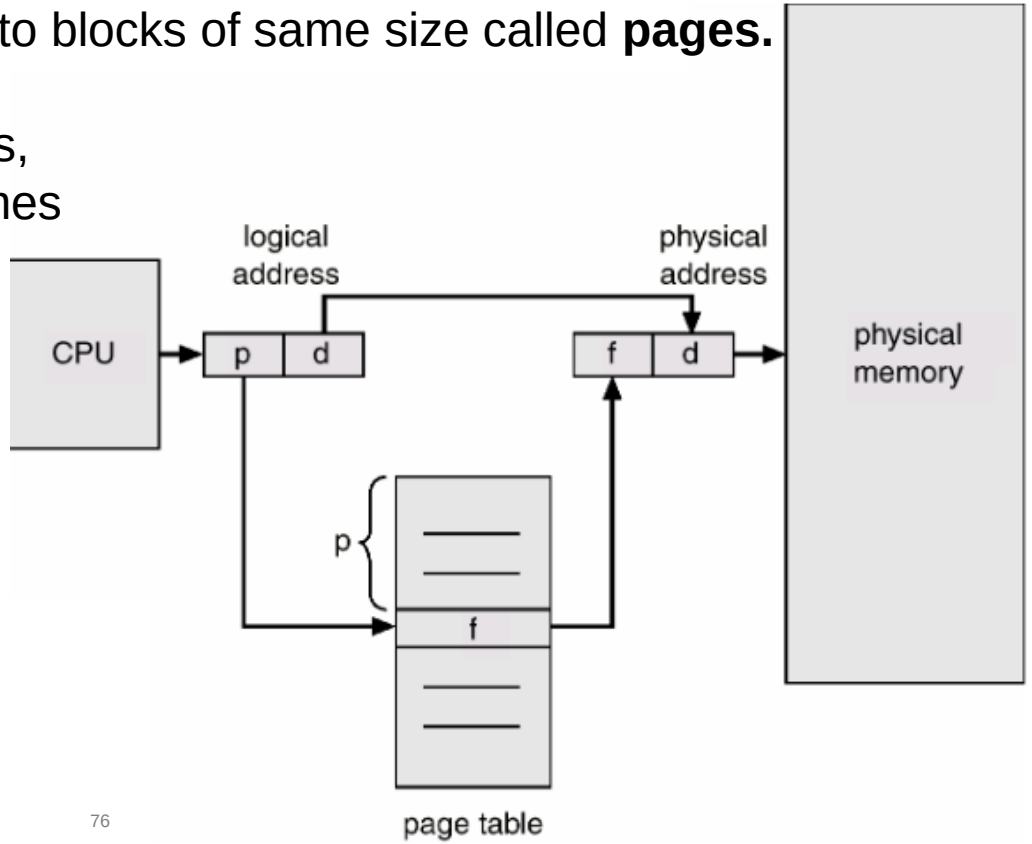
ltsirov@visteon.com

Concepts of OS: Memory management

Visteon

Logical address spaces of a process can be noncontiguous – process is allocating physical memory whenever that memory is available and the program needs it.

- The physical memory is divided into fixed sized blocks called **frames** (size is power of 2, between 512 and 8192 bytes)
- The logical memory is divided into blocks of same size called **pages.**
- Keep track of all free frames
- To run a program of size n pages, the OS needs to find n free frames and load the program
- **Page table** is used to translate logical to physical addresses



ltsirov@visteon.com

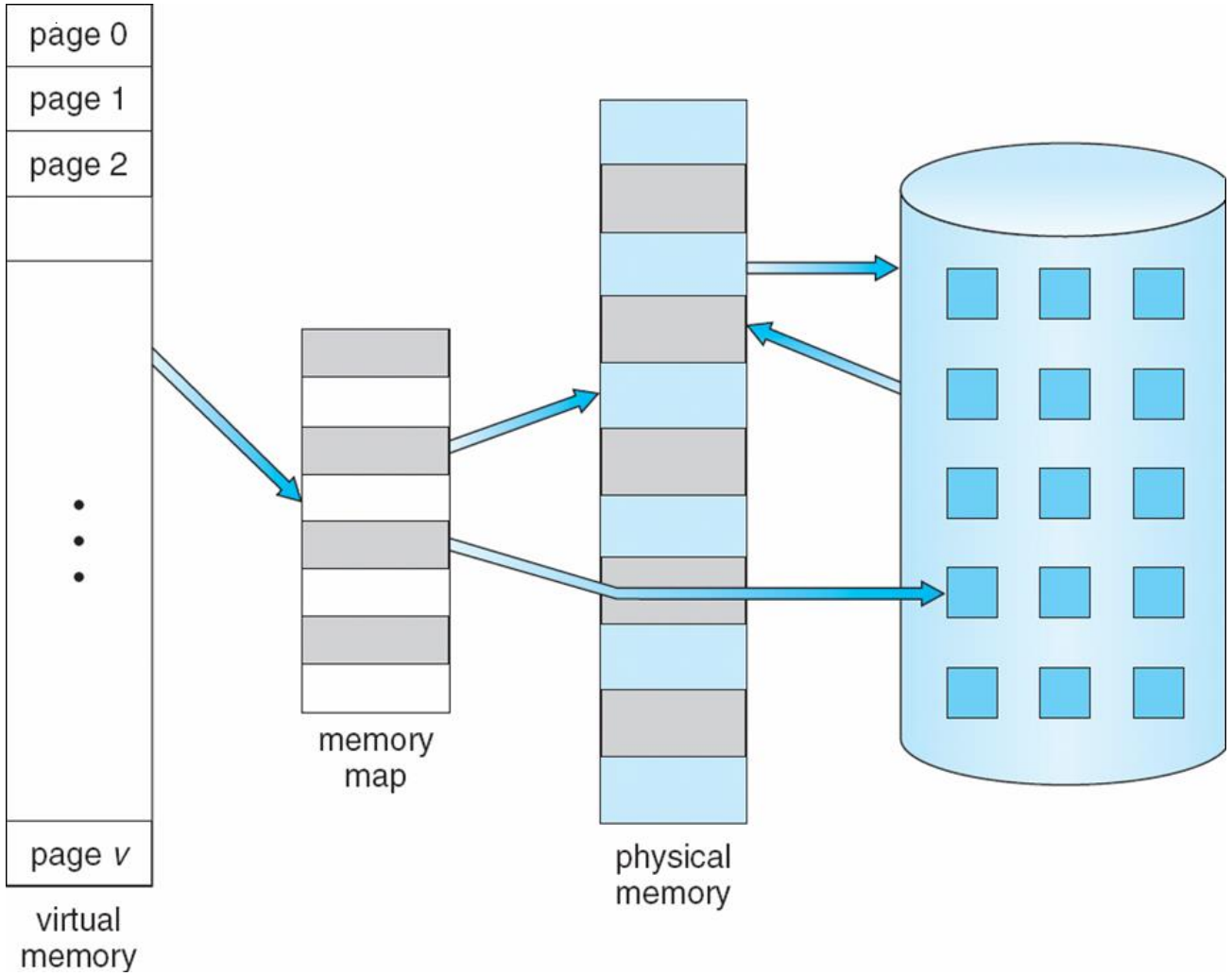
Visteon

Virtual memory addressing

- The address space (apparently accessible memory) of a process does not need to be identical to the physical address space.
- The virtual address space can be much larger than the physical space.
- Each process can have the same view of memory, for example, its program code is at (the same) very low virtual addresses even though physically this would be possible for at most one process.
- Program uses virtual addresses which the H/W maps to real store locations

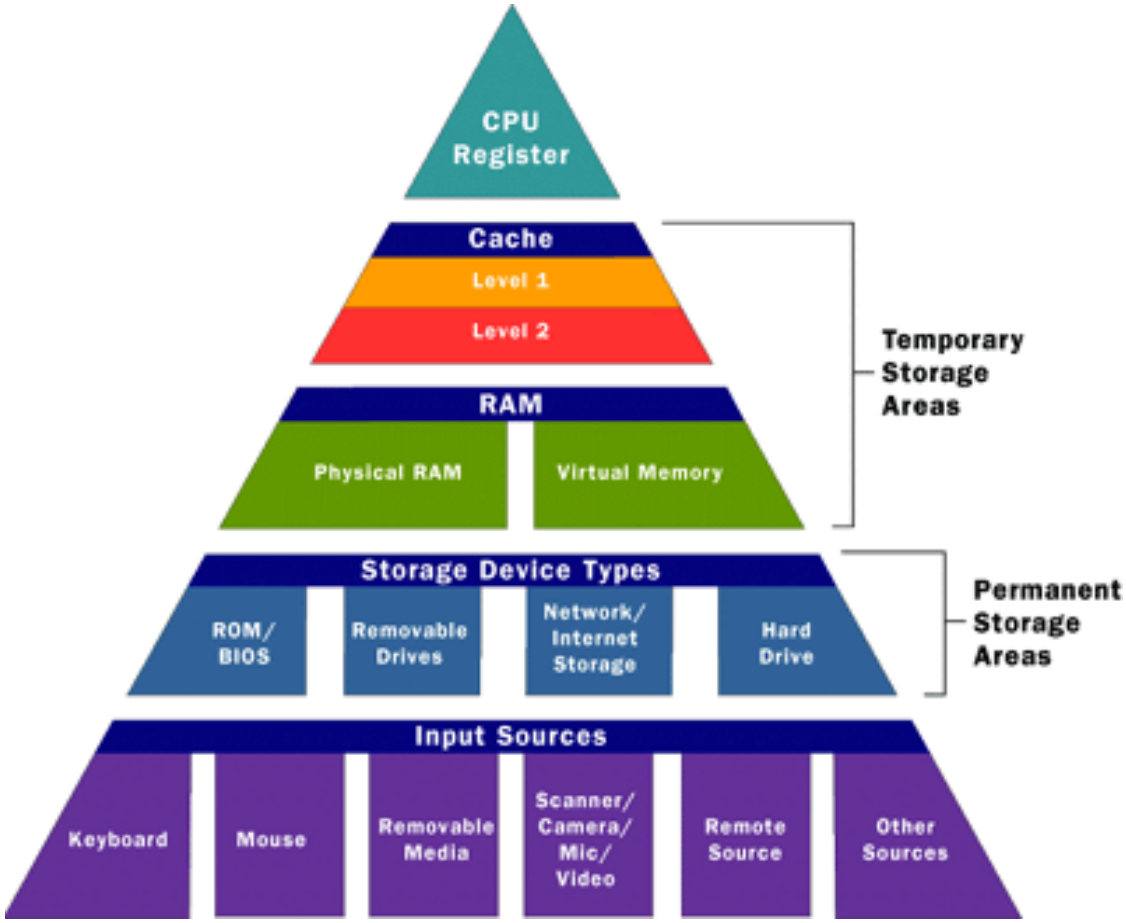
ltsirov@visteon.com

Concepts of OS: Memory management



ltsirov@visteon.com

Concepts of OS: Memory management

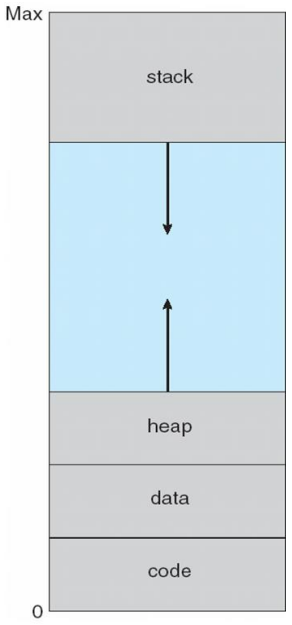


ltsirov@visteon.com

Concepts of OS: Memory management

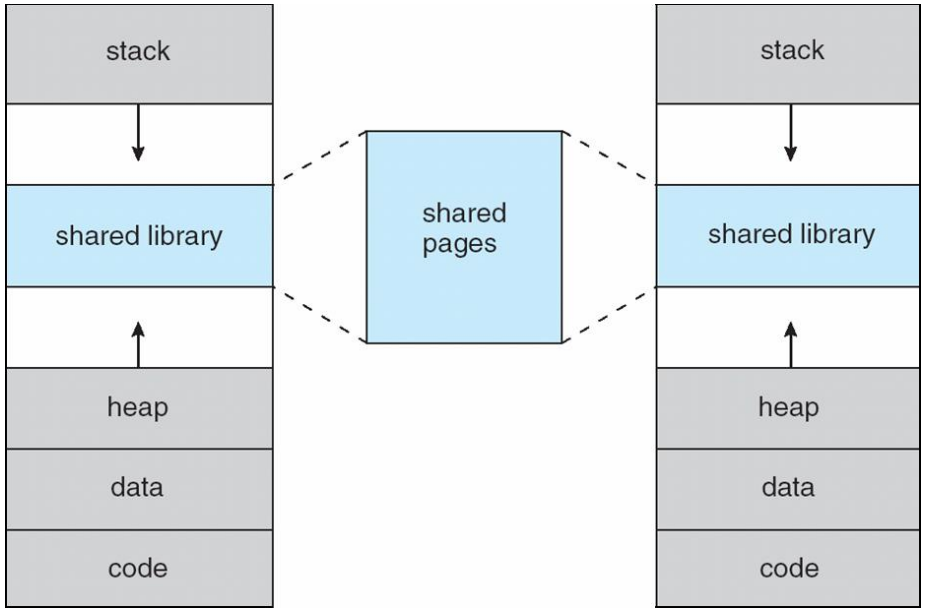
Visteon

Single process’s memory map



Multiple processes' memory map

- Pages can be shared



ltsirov@visteon.com

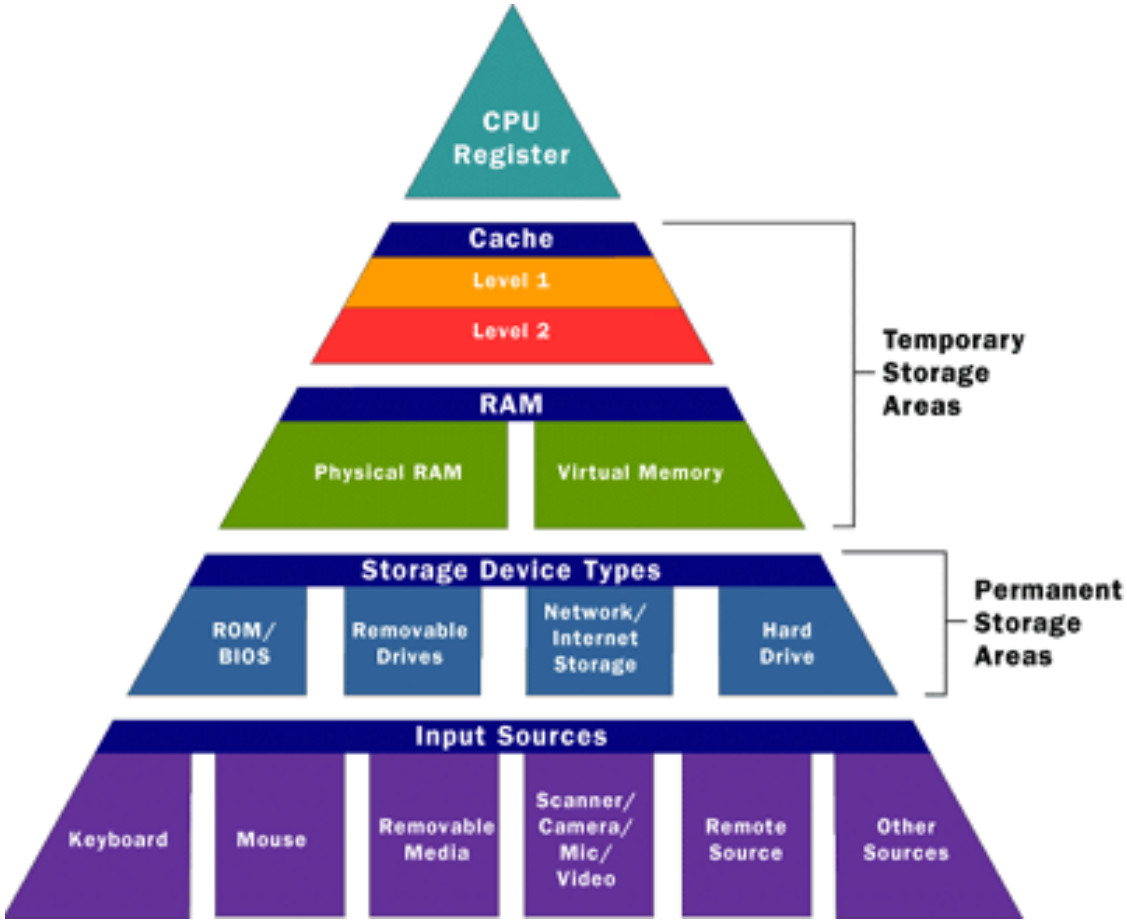
Contents

Visteon

1. What is an operating system?
2. Kernel
3. System call
4. Concurrency – issues and solutions
5. Multi process programming
6. Multi thread programming
7. Concepts of OS
   - Scheduling(time sharing/time slicing)
   - Memory management & Virtual memory(address space)
   - **I/O access**
   - **Interrupts**
   - **Real time operations**
8. Inter process communication in UNIX

ltsirov@visteon.com

Concepts of OS: I/O Access



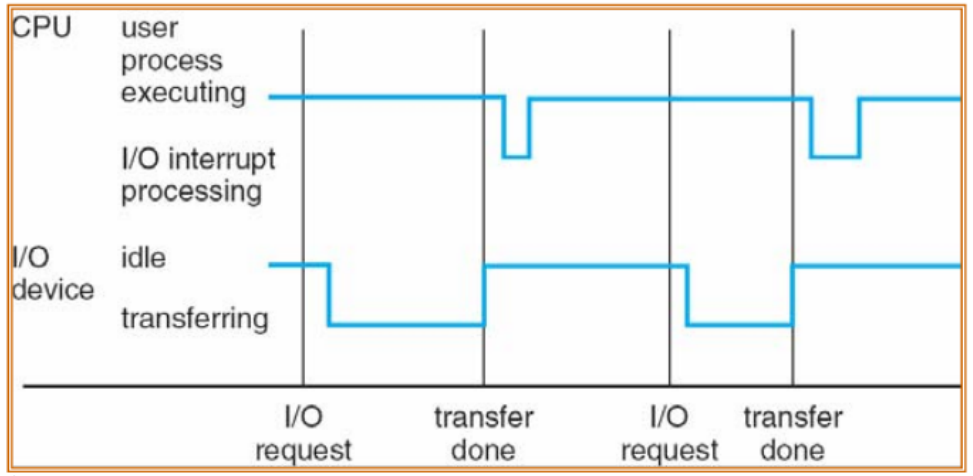
ltsirov@visteon.com

Concepts of OS: I/O Access

I/O access is generally slow, but most of the programs are depended on I/O devices.
How do we know that I/O operation is complete?

- Polling:
  - I/O device sets a flag when it is busy.
  - Program tests the flag in a loop waiting for completion of I/O operation.

- Interrupts:
  - The I/O operation is handled asynchronously
  - When a program performs I/O, instead of polling, another program can be executed till interrupt is received.



ltsirov@visteon.com

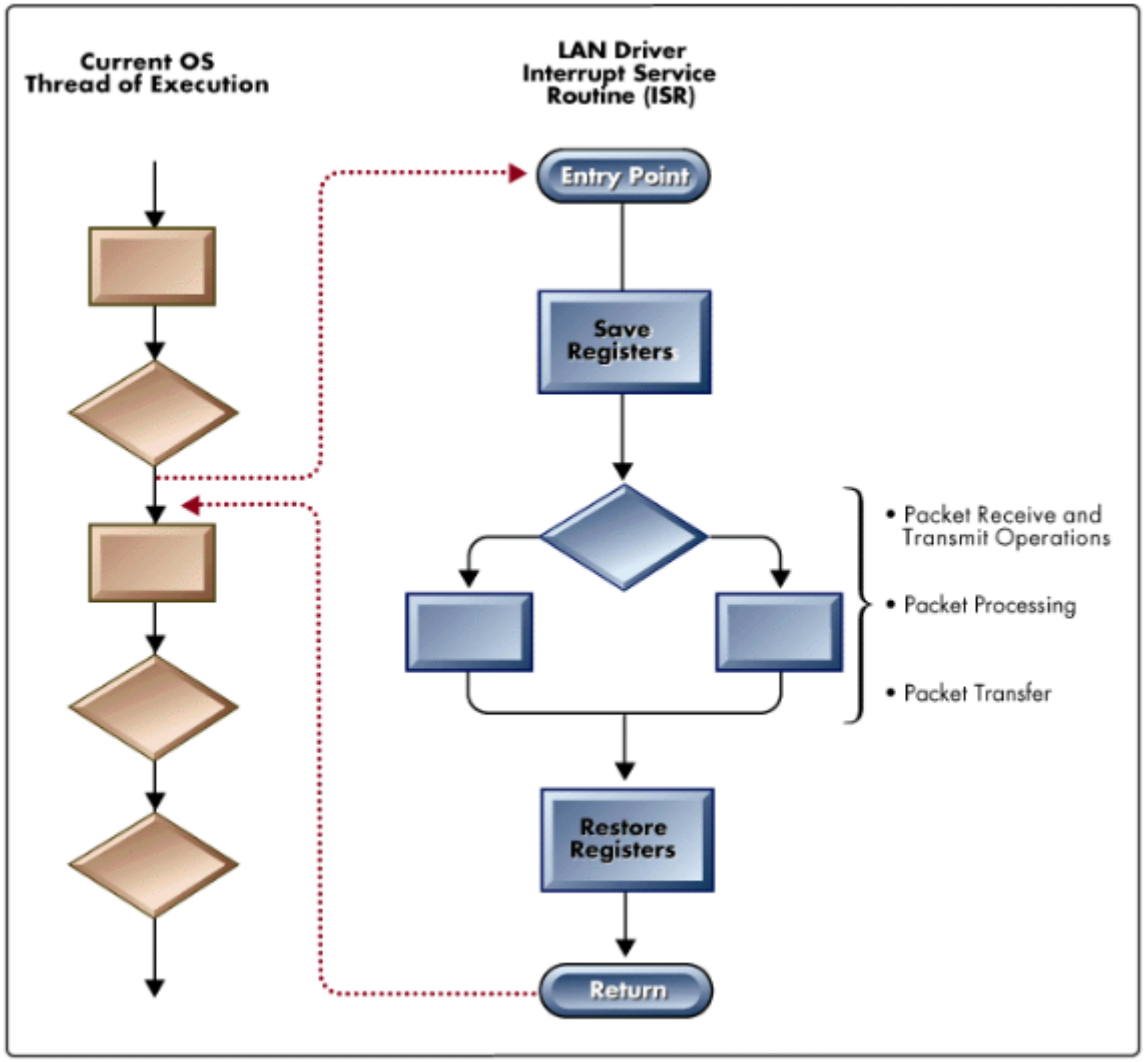
Concepts of OS: Interrupts

Visteon

- On completion of I/O transfer or other external event, an interrupt is triggered and it forces CPU to jump to a specific location in the memory contains the interrupt service routine.
- After the interrupt has been processed, CPU returns to code it was executing prior to servicing the interrupt.
- The operating system preserves the state of the CPU by storing registers and the program counter.
- Incoming interrupts are disabled while another interrupt is being processed to prevent a lost interrupt (there are some exceptions though)
- A trap is a software-generated interrupt caused either by an error(e.g. division by zero) or a user request (e.g. request for operating system service).
- An operating system is interrupt driven.

ltsirov@visteon.com

Concepts of OS: Interrupts



http://support.novell.com/techcenter/articles/img/ana1995050101.gif

ltsirov@visteon.com

Concepts of OS: Real time operations

Visteon

- Real time in operating systems:

  "*The ability of the operating system to provide a required level of service in a bounded response time.*"

  POSIX standard 1003.1

- Real time handling of interrupts and user actions
  - All inseparable program segments(including interrupt handlers themselves) must be completed within a well-defined fixed-time constraints.

ltsirov@visteon.com

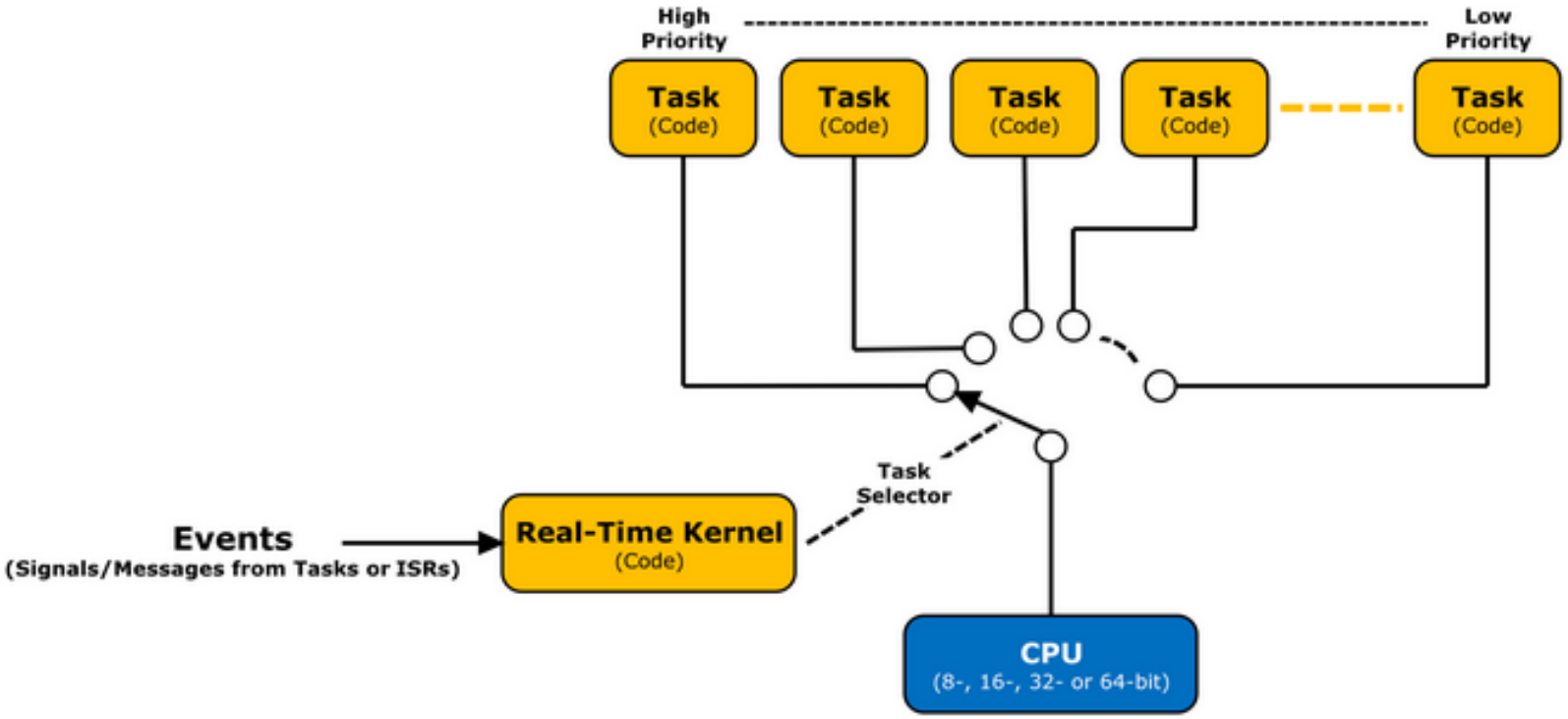
Concepts of OS: Real time operations

Two types of real time systems:

- Hard real time systems
  - Failure if response time too long.
- Soft real time systems
  - Less accurate if response time is too long

ltsirov@visteon.com

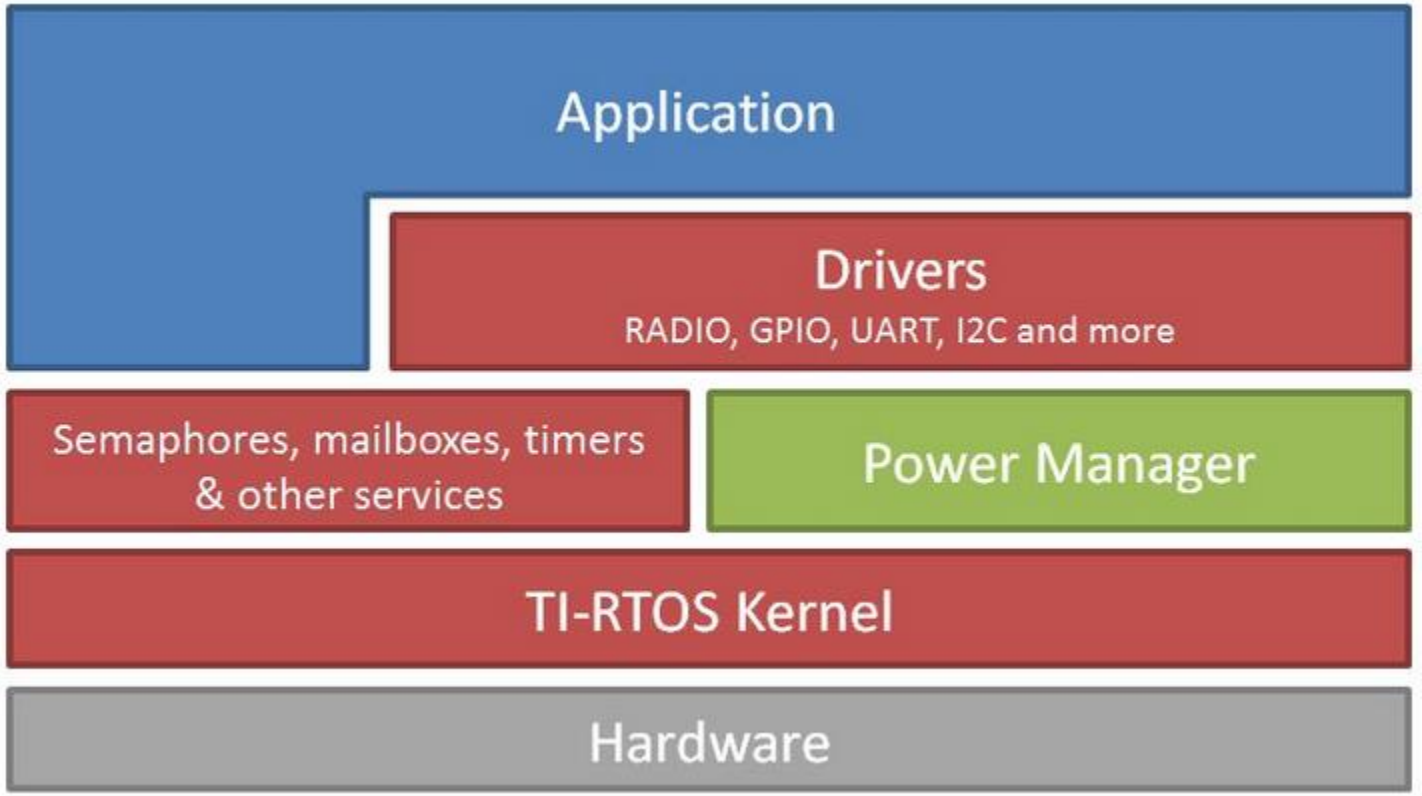
Concepts of OS: Real time operating systems



ltsirov@visteon.com

Concepts of OS: Real time operating systems

Visteon

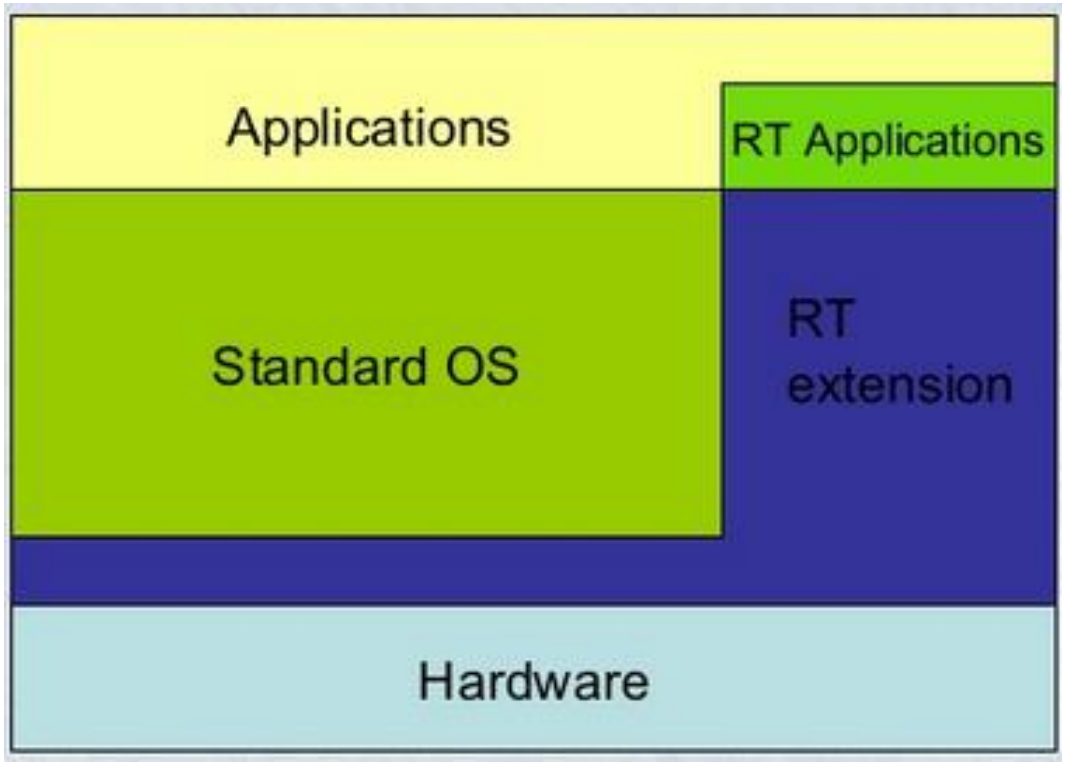


ltsirov@visteon.com

Concepts of OS: Real time operating systems

Visteon

Automotive electronics is a real time environment and real time operating systems(RTOS) are widely adopted.



ltsirov@visteon.com

Contents

1. What is an operating system?
2. Kernel
3. System call
4. Concurrency – issues and solutions
5. Multi process programming
6. Multi thread programming
7. Concepts of OS
   - Scheduling(time sharing/time slicing)
   - Memory management & Virtual memory(address space)
   - **I/O access**
   - **Interrupts**
   - **Real time operations**
8. Inter process communication in UNIX

ltsirov@visteon.com

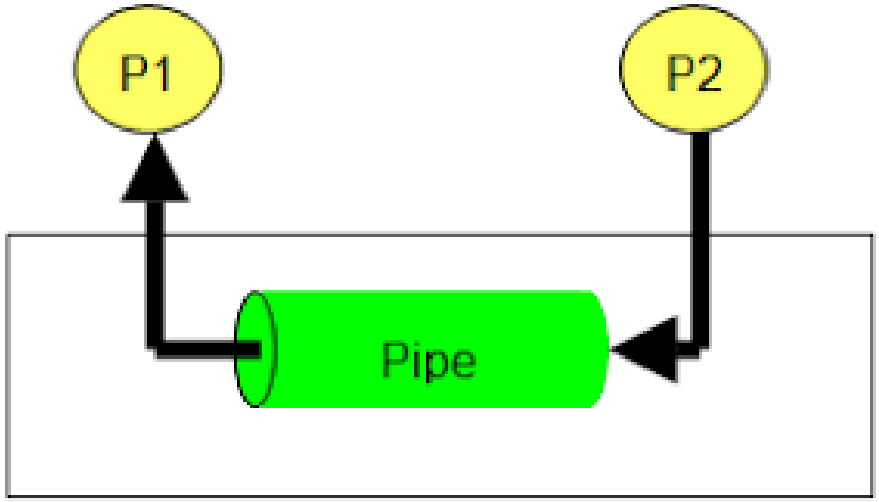
Inter process communication in UNIX

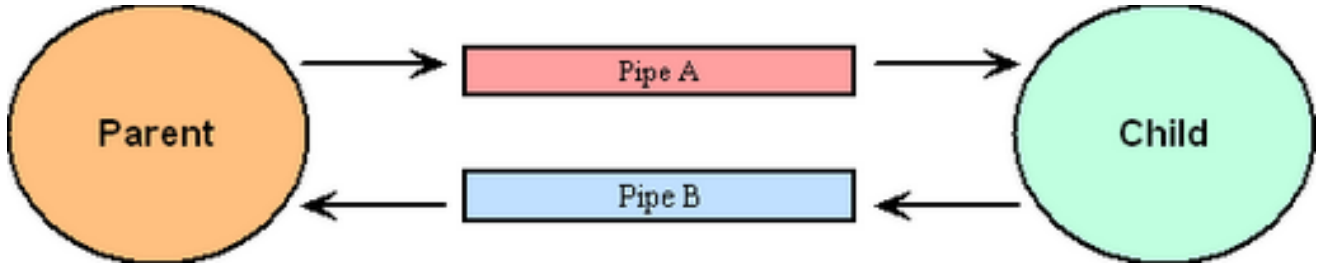
Visteon

- Linux Files and system functions
- Pipes
- Sockets (UNIX & TCP/IP)
- Message queues
- Shared memory

ltsirov@visteon.com

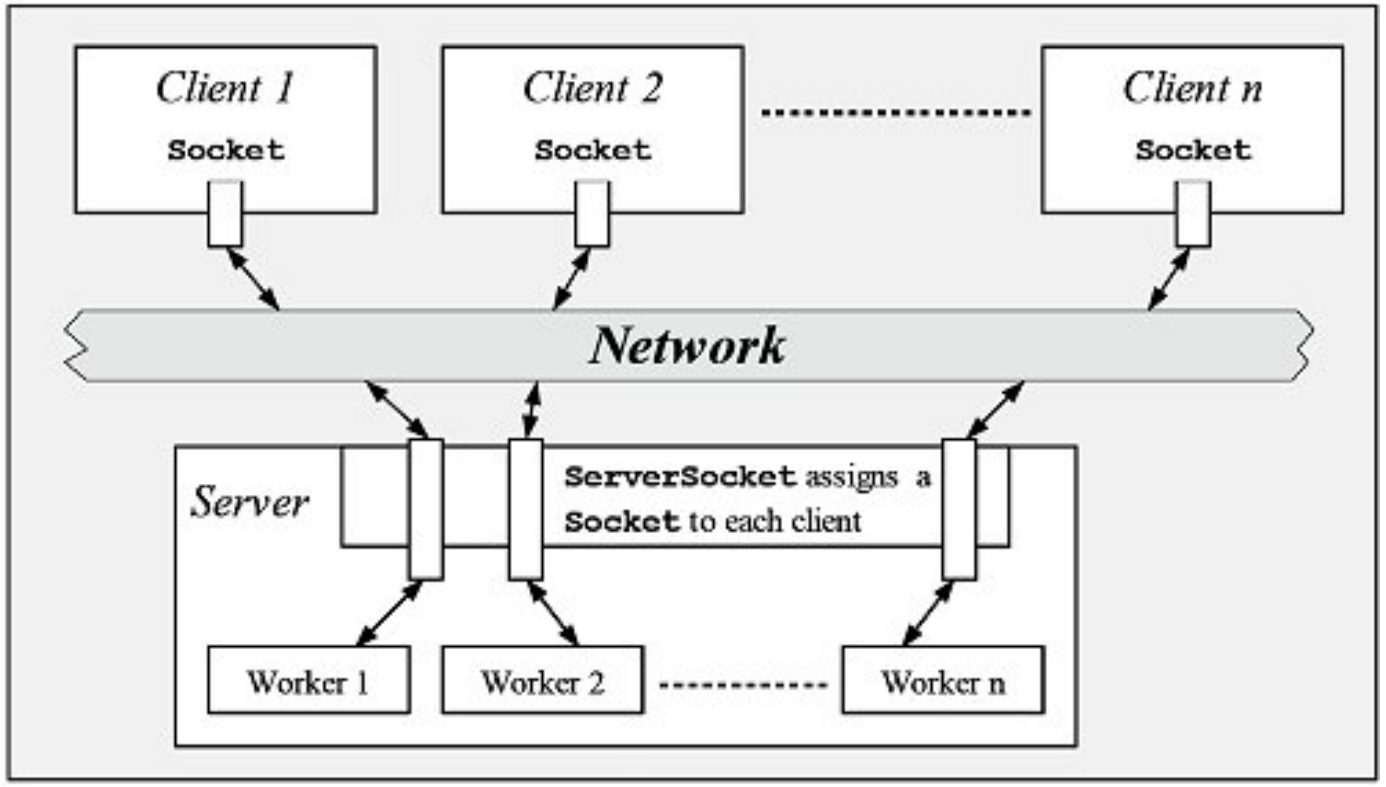
Inter process communication in UNIX - pipes





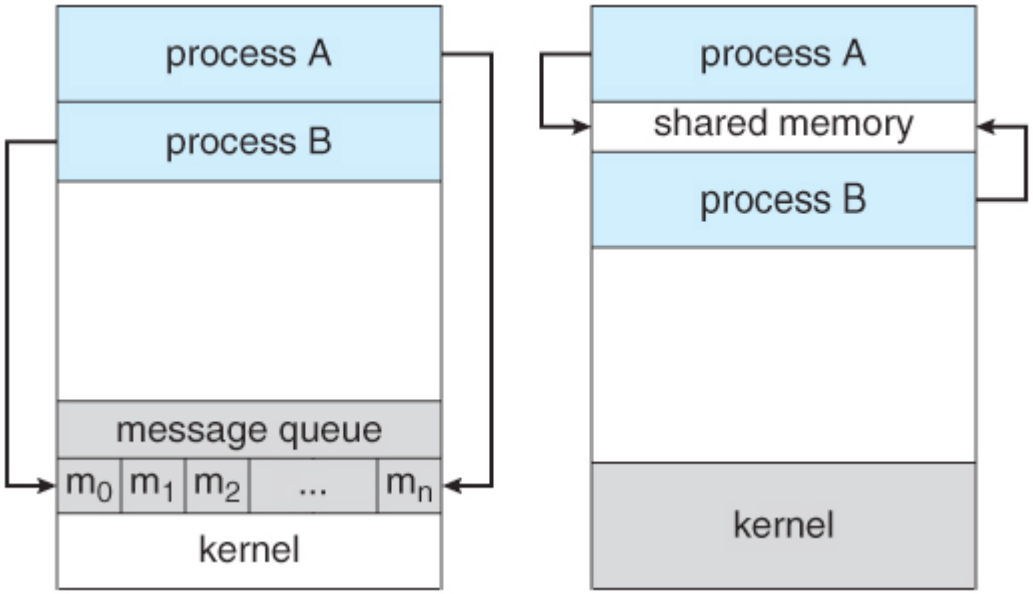
ltsirov@visteon.com

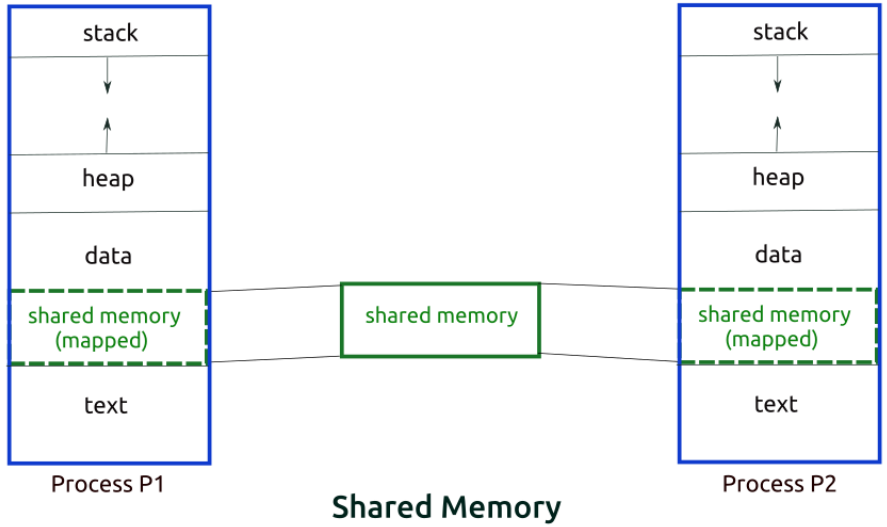
Inter process communication in UNIX - sockets



ltsirov@visteon.com

Inter process communication in UNIX – mqueue, shared memory





ltsirov@visteon.com

Visteon®

ltsirov@visteon.com

Useful references

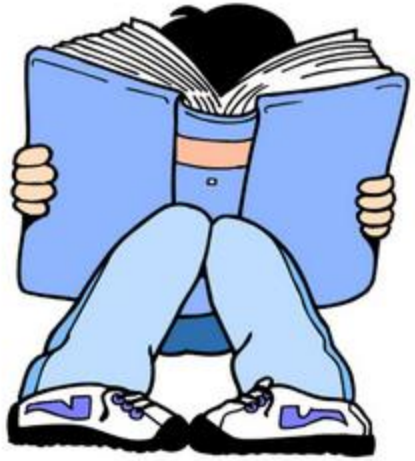
Visteon

Literature

- The GNU C Library Manual
  - PDF: htttps://www.gnu.org/software/libc/manual/pdf/libc.pdf
  - HTML: https://www.gnu.org/software/libc/manual/html_node/

•Books

- Linux System Programming: Talking Directly to the Kernel and C Library – Robert M. Love
- Modern Operating Systems – Andrew S Tanenbaum
- The Linux Programming Interface - Michael Kerrisk
- Advanced Linux Programming - Richard Esplin



ltsirov@visteon.com