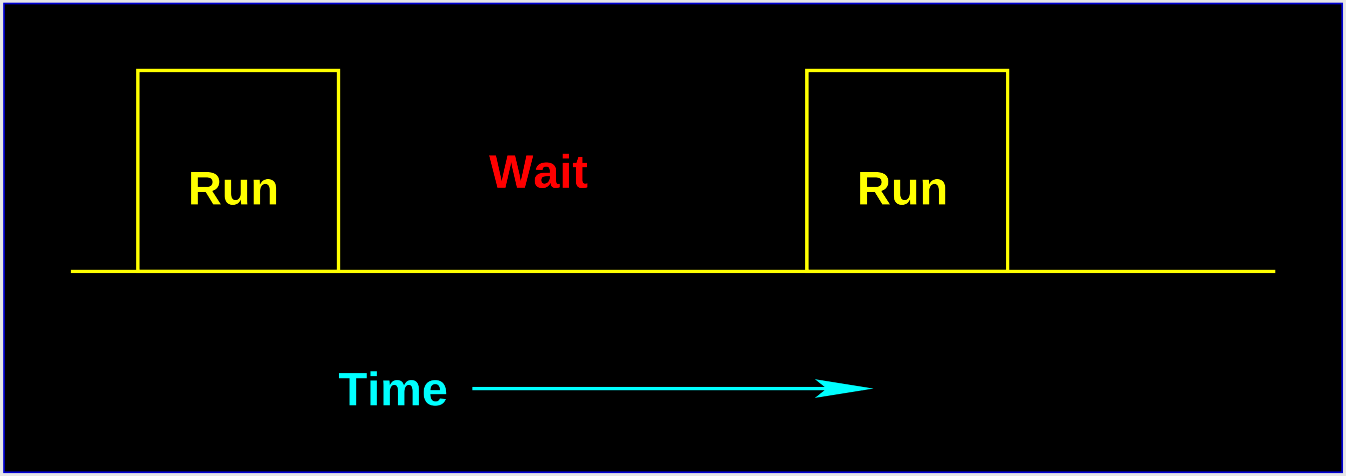
Multitasking and Concurrency - 1
Review of theoretical background

Prof. Himanshu B. Dave
e-Infochips, Ahmedabad

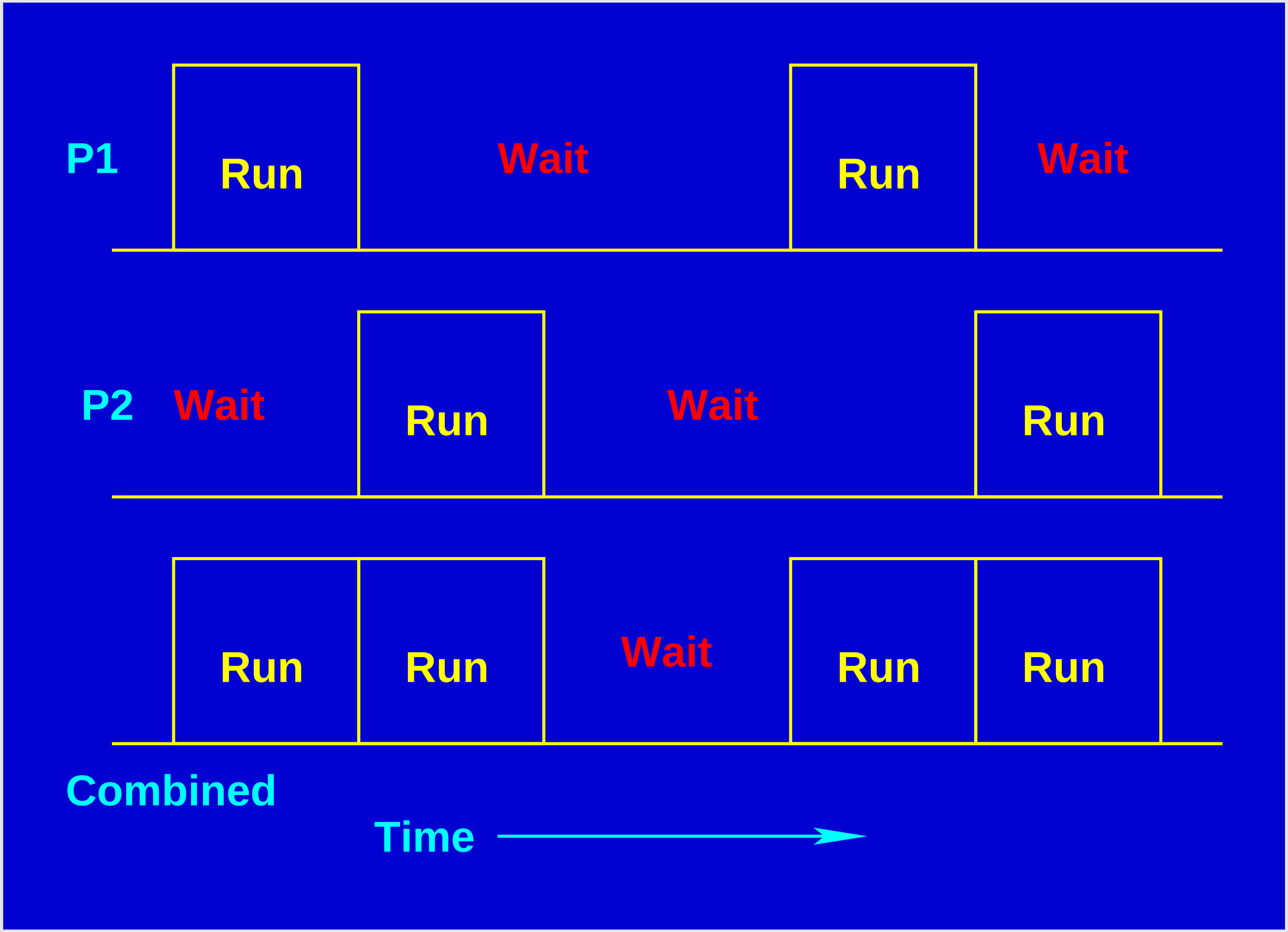
June, 2010 / e-Infochips, Ahmedabad

Contents

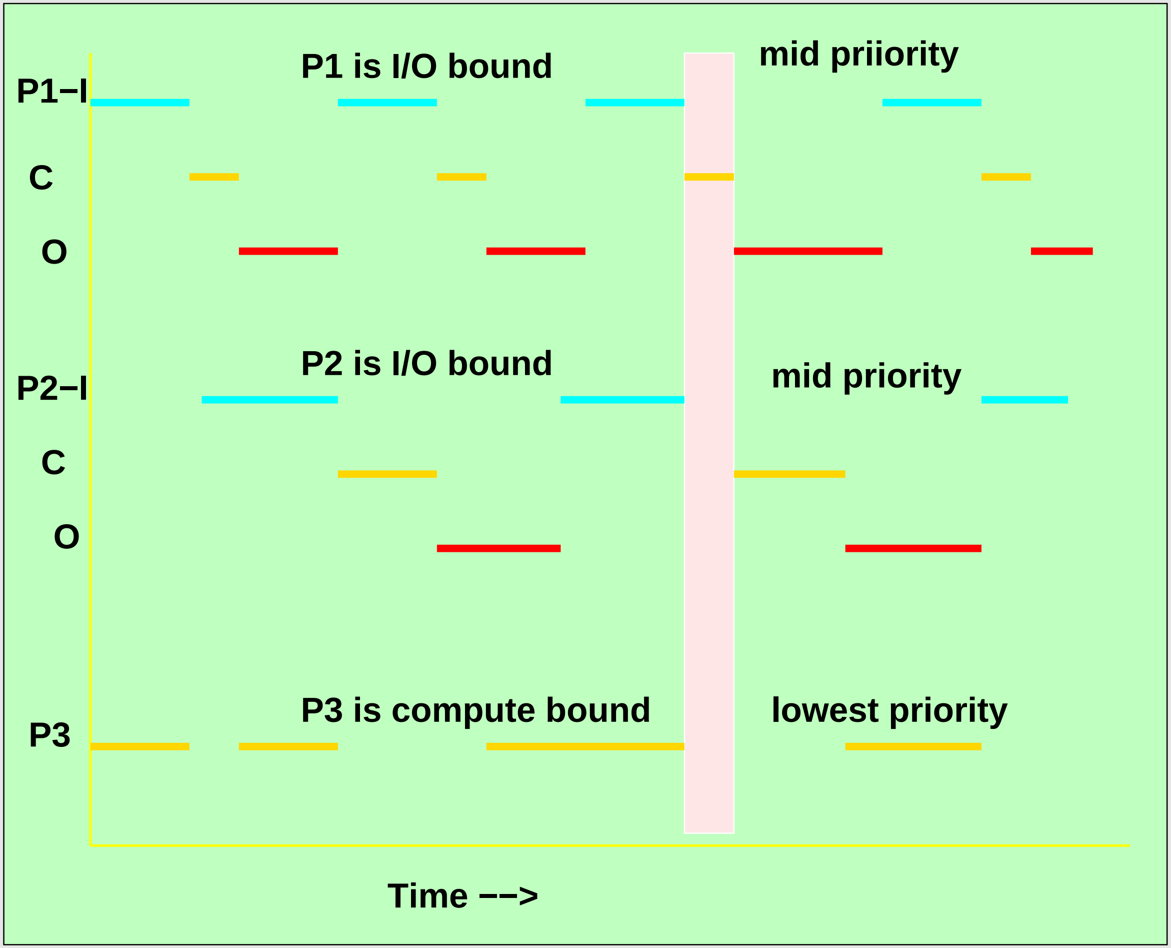
Uniprogramming



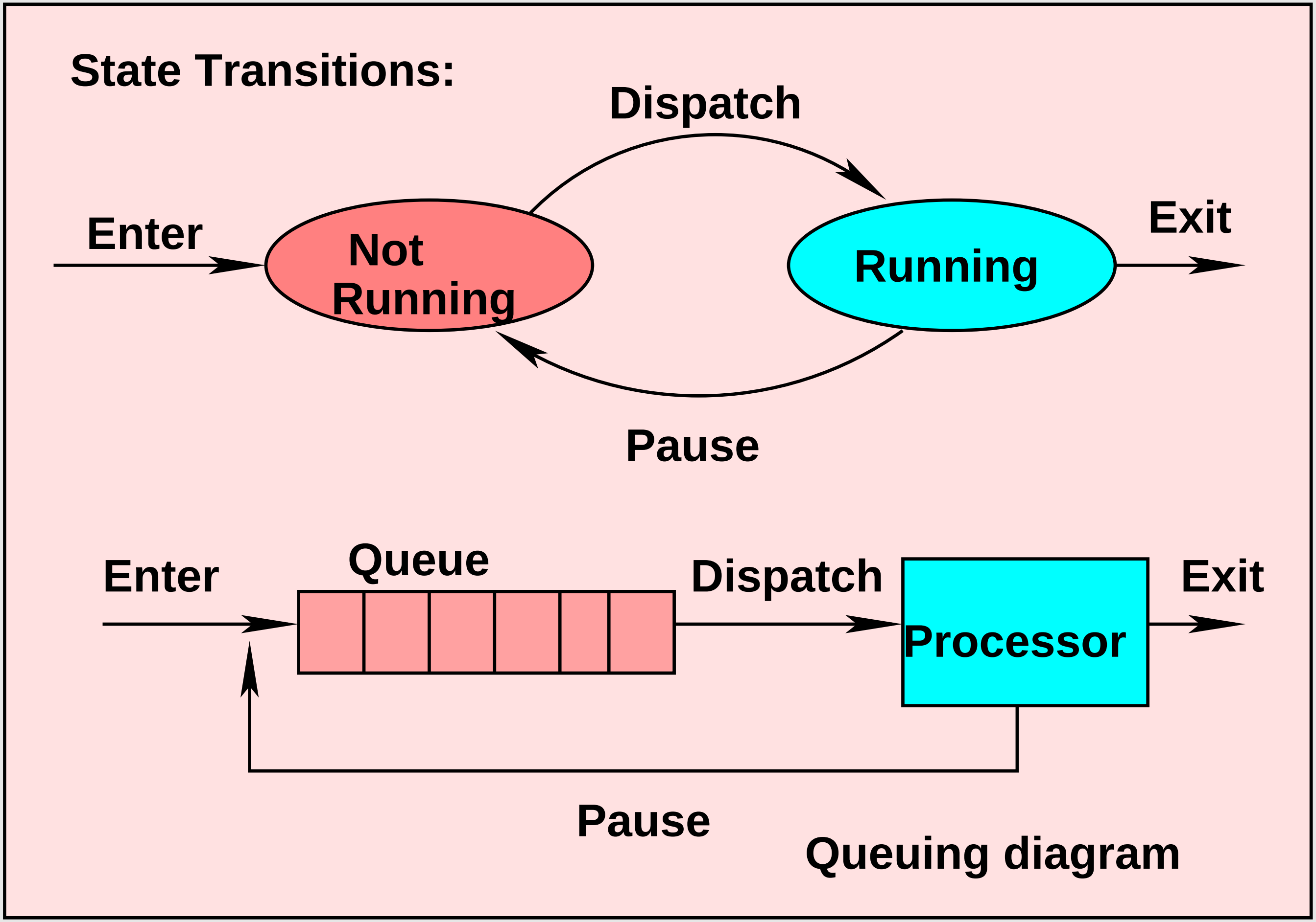
Multi-programming



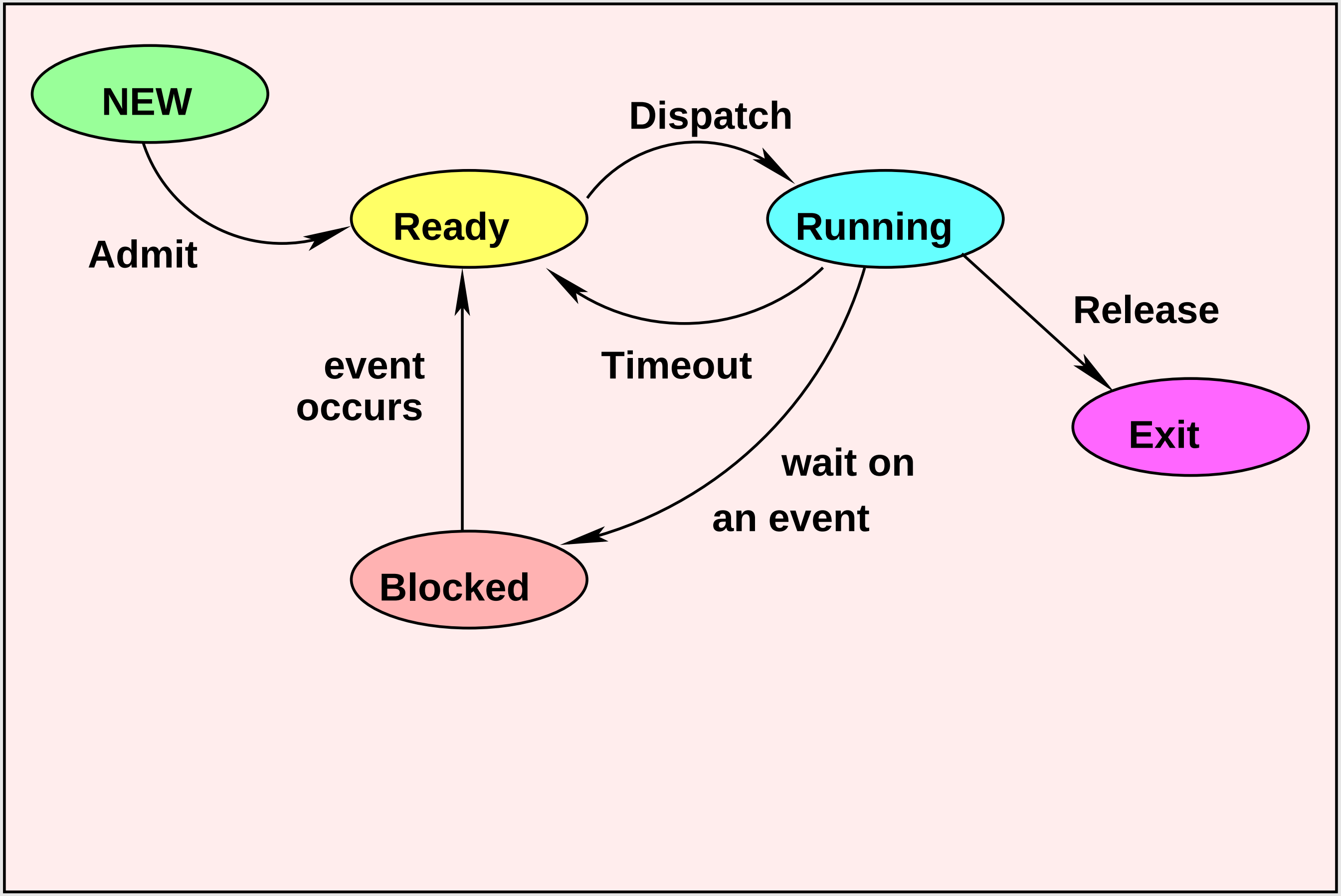
Several Tasks executing on a Single Processor



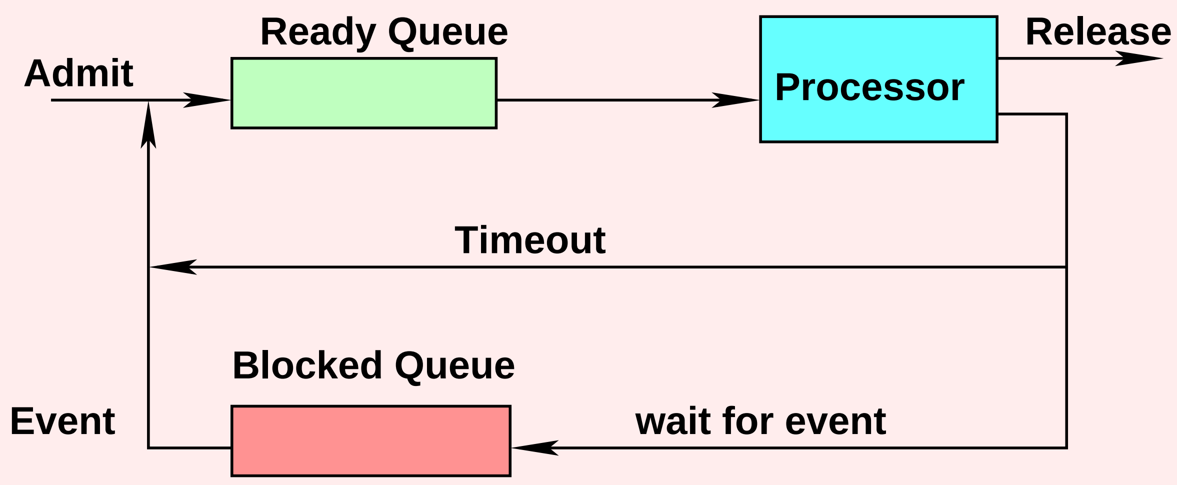
Two-state model of a Process



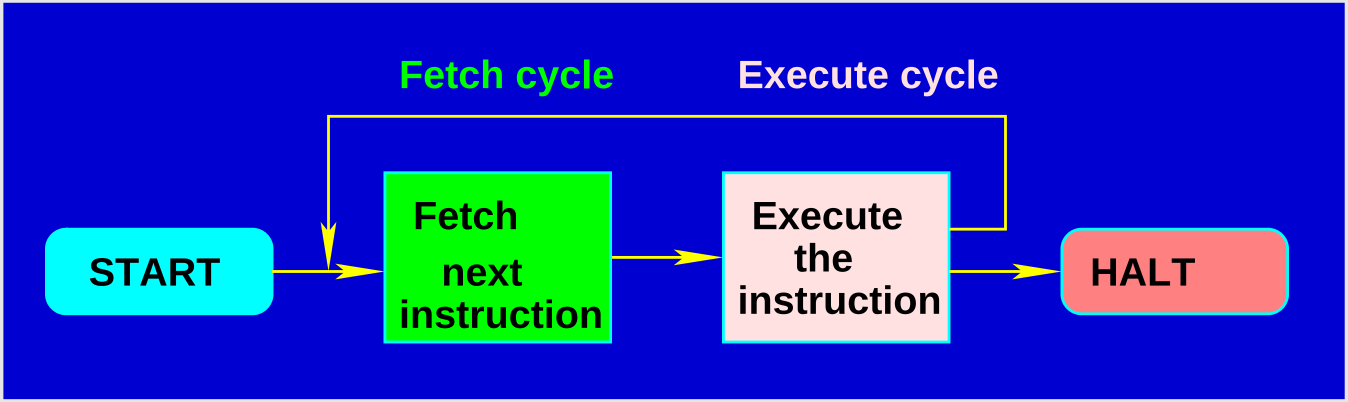
Five-state model of a Process



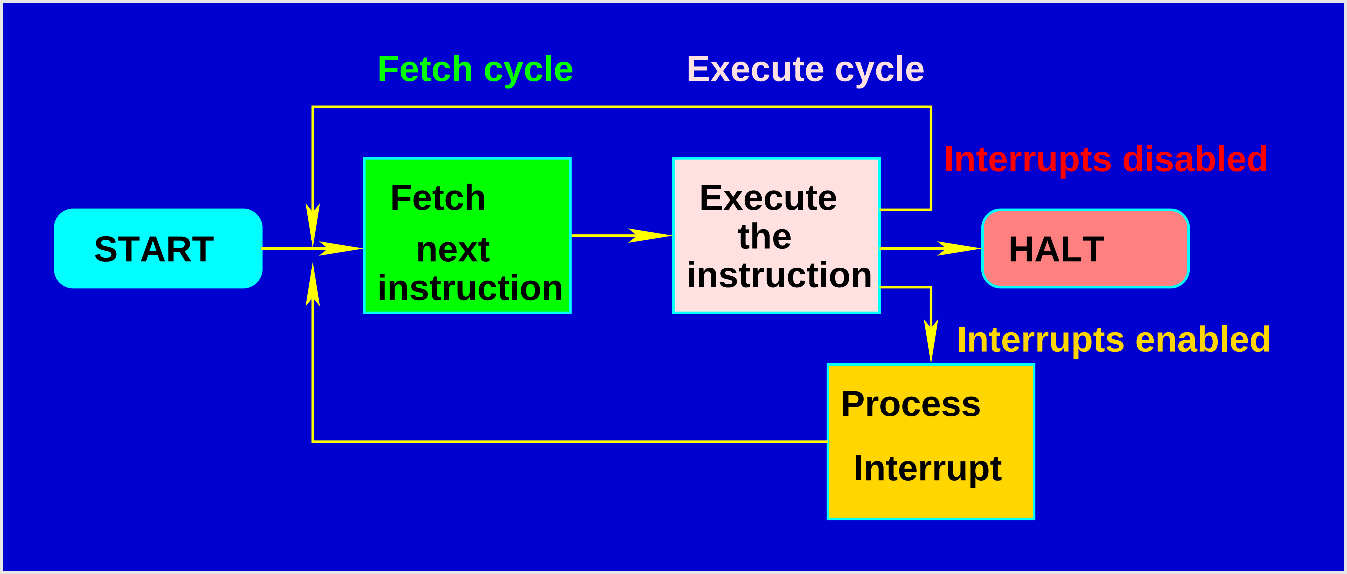
Queuing model for five states of a Process



How atomicity is created: F/E cycle



How atomicity is created: Interrupt cycle



Concurrent Processing

What is Concurrency?

- "sharing of resources in the same time frame".
- several processes share the same CPU, memory or I/O devices;
- requires correct co-ordination of processes;
- previously O/S handeled concurrency,
- no more true, now programmers have also to worry about concurrency:

  – complex applications,
  – multi-processor architecture,
  – distributed systems;

programming to achieve and control concurrency

Concurrency

- A sequential program has a **single** *thread of control* or execution-context. Its execution is called a process.
- A concurrent program has **multiple** *threads of control*, or execution-contexts. These *may be* executed as parallel processes.

Concurrency

What is "going on at the same time"?

processes progress to their completion at the same time

Is it parallelism?

No: parallelism would mean instant by instant, simultaneous operations by more than one processors

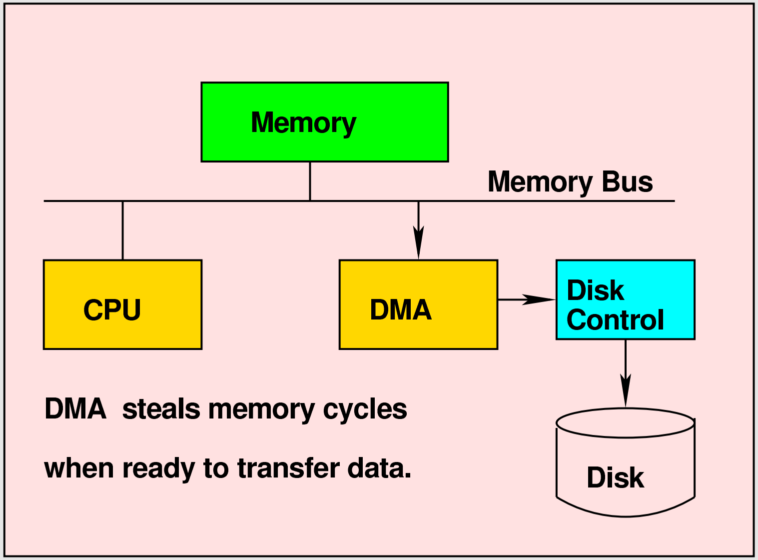


Figure 1

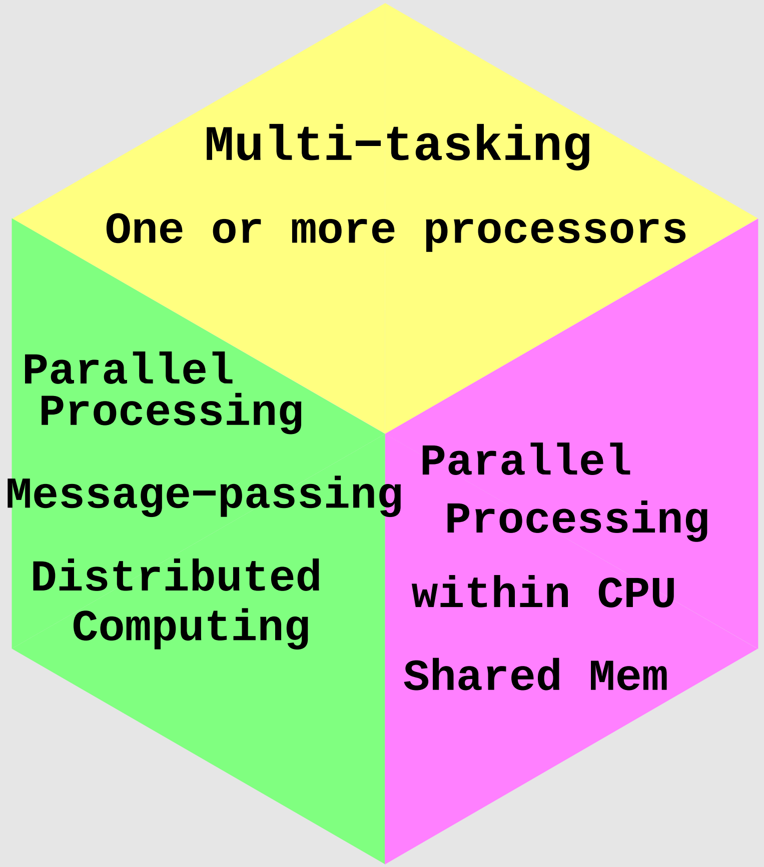
Concurrency and Parallelism

A concurrent program can be executed by:

- Multiprogramming: processes share *one* or more processors
- Multiprocessing: each process runs on its own processor but with shared memory
- Distributed processing: each process runs on its own processor connected by a communication network to others

Assume only that all processes make positive finite progress.

Concurrency can be found in:



Concurrency

Multi-processing: bus with shared memory

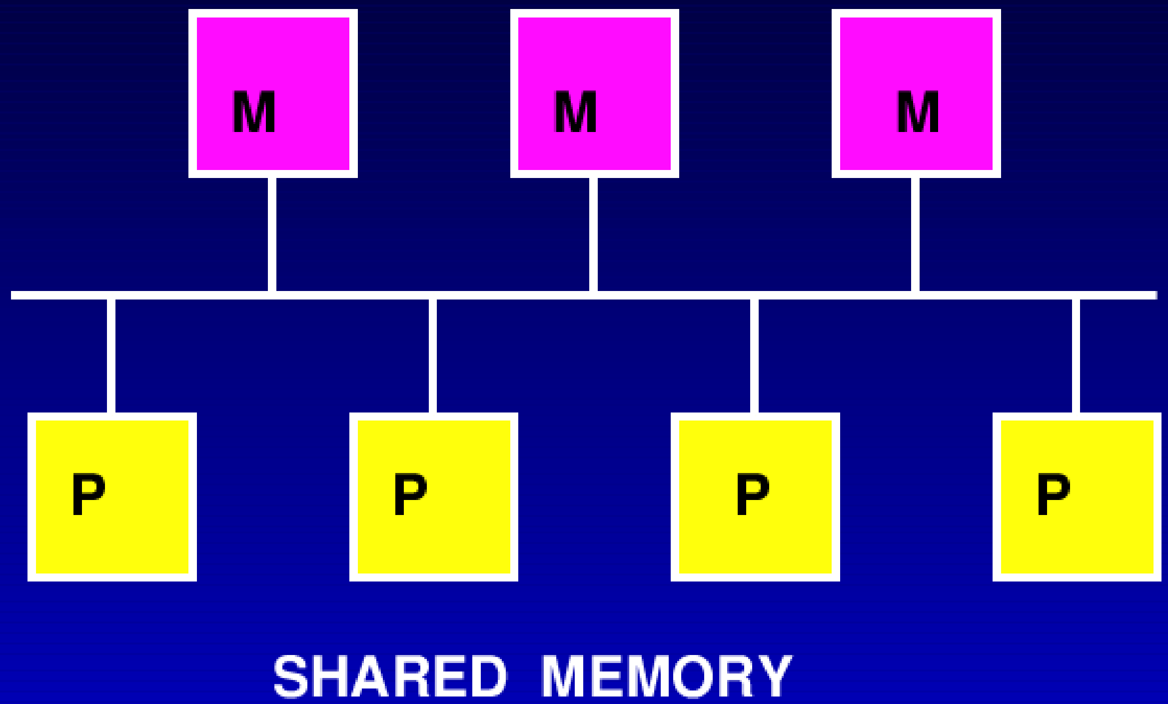


Figure 1: Bus-based, Shared Memory

Multi-processing: bus with private memory

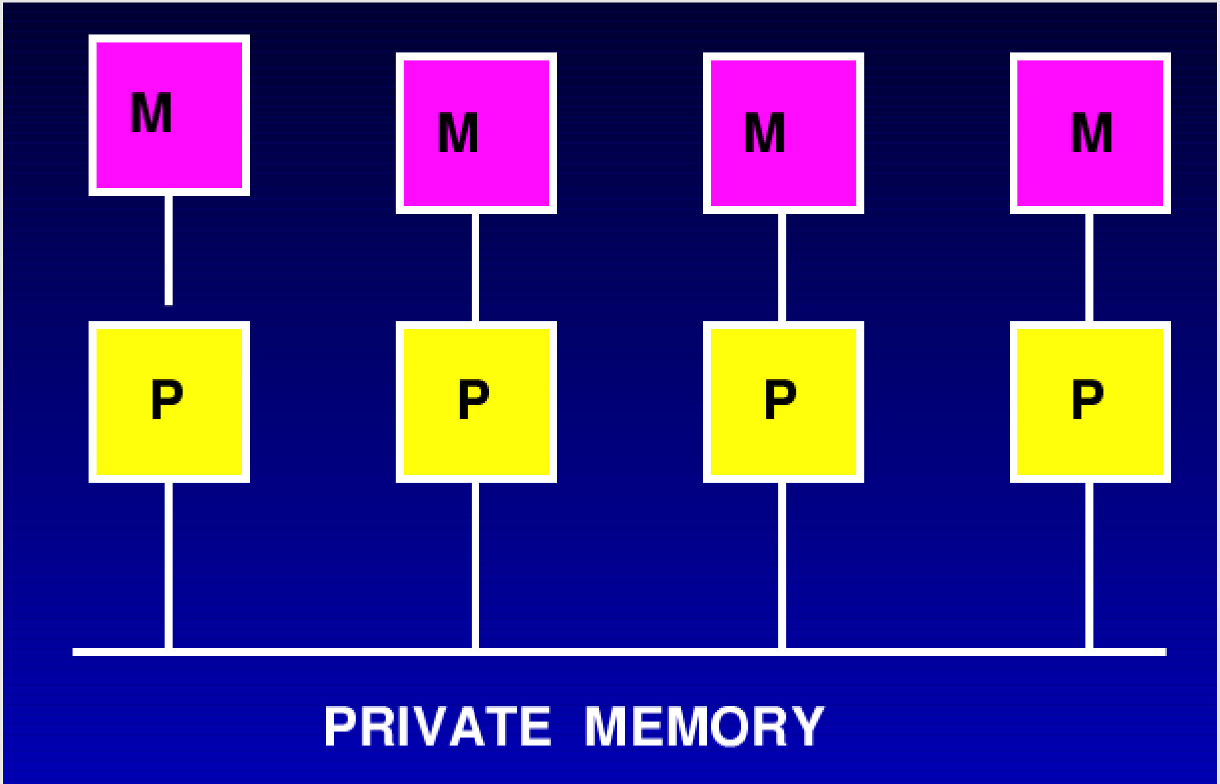


Figure 2: Bus-based, Private Memory

Multi-processing: independent CPU’s on bus

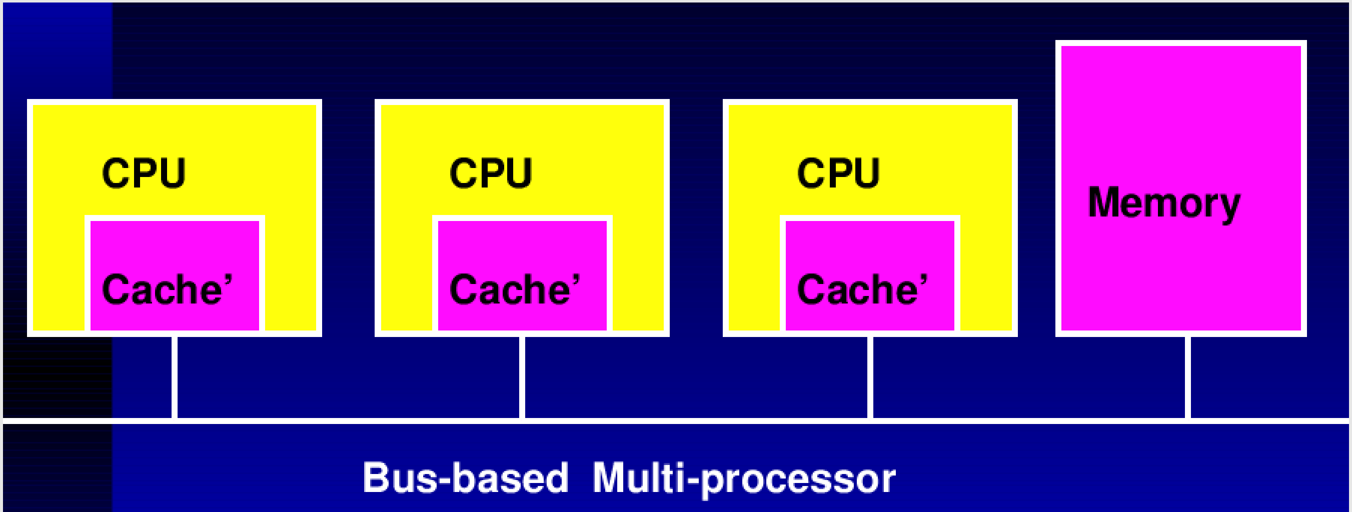
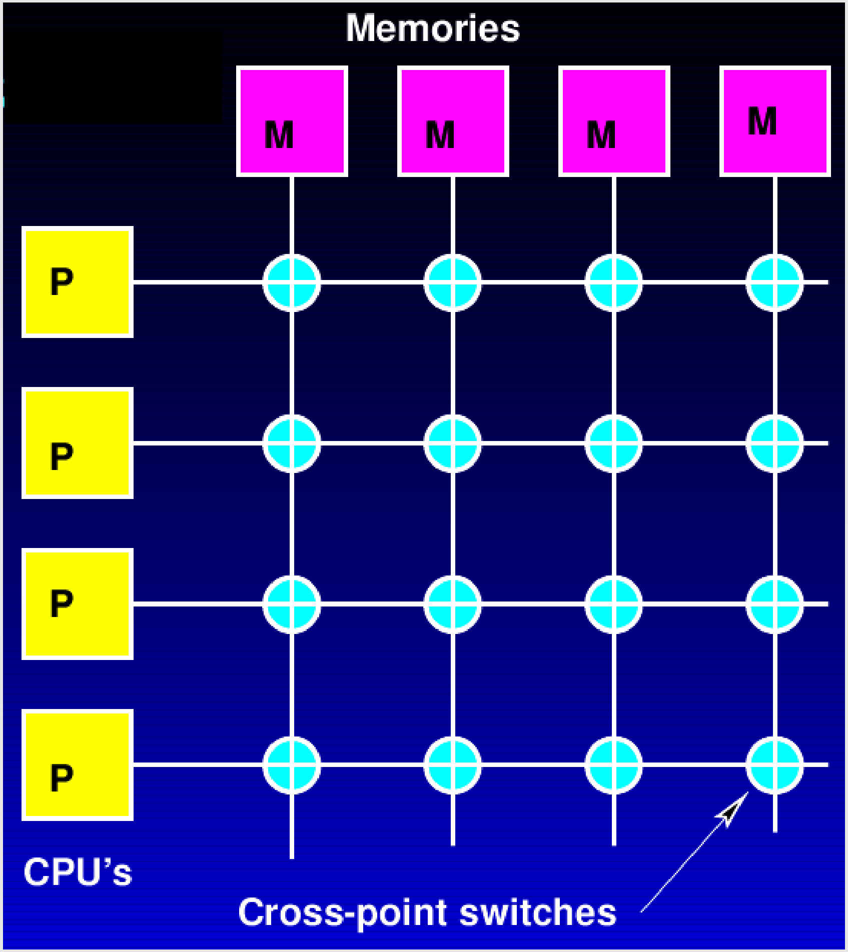


Figure 3: A Bus-based Multiprocessor

Multi-processing: PU and MEM connected by Cross-bar switch



Where and when is Concurrency needed?

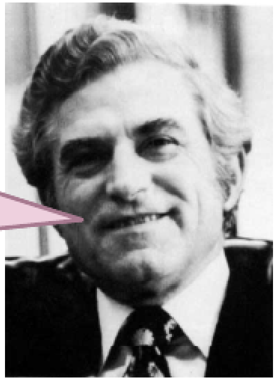
- logic of application demands it
- application is easier to visualize and implement that way;
- resource utilization issues demand it;
- speed-up execution by parallel operation on multiple processors, this has become a necessity as we reach limits of VLSI component size
- coordinate distributed services (e.g. Web-Services)

Amdahl's Law

Amdahl's Law

If 50% of your application is parallel and 50% is serial, you can't get more than a factor of 2 speedup, no matter how many processors it runs on.*

*In general, if a fraction α of an application can be run in parallel and the rest must run serially, the speedup is at most $1/(1-\alpha)$.



Gene M. Amdahl

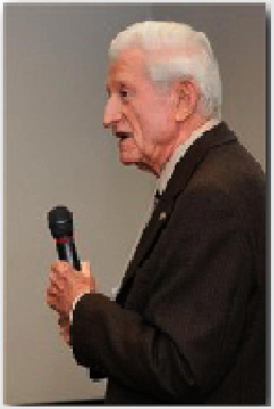
But, whose application can be decomposed into just a serial part and a parallel part? For *my* application, what speedup should I expect?

Two Pioneer Computer Enginers

Gordon E. Moore

Gene Amdahl

No, they're not actors



Moore’s and Amdahl’s Laws

- **Moore's Law.**
  - *The number of transistors that can be inexpensively placed on an integrated circuit is increasing exponentially.*
  - Not true anymore!
- **Amdahl's Law.**
  - *Performance decreases as number of processors increases once there is even a small percentage of non-parallelizable code.*
  - This is our new reality!

Problem of Growing!

- We live in the **multi-core/multi-processor era**.
- But we're not prepared for it ...
  - Most of our software is **non-parallelizable**.
  - Most of our software is written for **single-processor**.
  - Most of our software has **shared state**.

Present day approach

- Shared state model.
  - The way we're used to.
    - We have a few variables.
    - We have one or more threads.
    - We have our threads accessing our variables.
    - We have to acquire/release **locks**.
      - ✓ The right locks.
      - ✓ In the right order.
      - ✓ For the right resources.

Difficulties with concurrent execution

Concurrency introduces complexity:

- concurrent processes may corrupt shared data (Safety)
- processes may "starve" if not properly coordinated (Liveness)
- the same program run twice may give different results (Non-determinism)
- thread construction, context switching and synchronization take time (Run-time overhead)

Concurrency by creating new processes

Remember C library function fork() ?

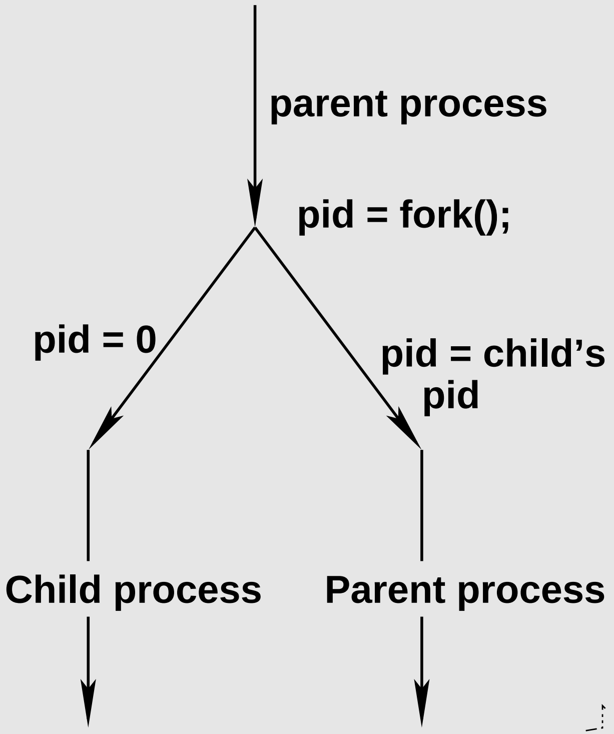


Figure 2

Threads

Consider a file server that has several *"Threads of Control"* or *"Context of Execution"* as Linus calls it.

- The server will block occasionally for disk I/O.
- While one thread is sleeping, another could continue its execution.
- Net result: higher throughput, better performance.

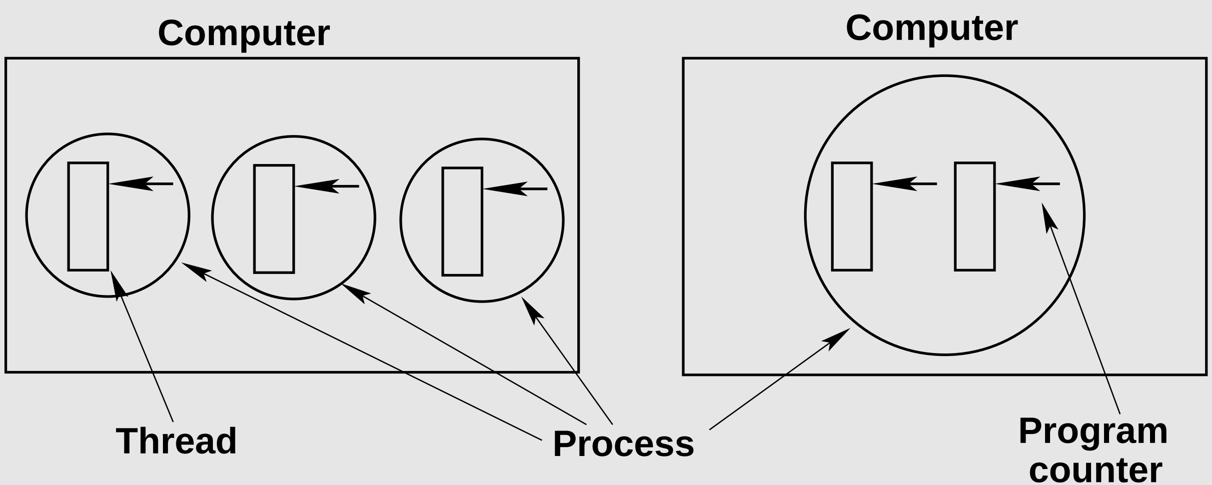


Figure 3

Sharing of Resources

- Two problems: Race conditions and Dead-locks
- co-ordination between Threads required:

– critical sections
– mutual exclusion

What is the value of x?

Execution-context P1 and P2 run concurrently:

- x $\leftarrow$ 0
- P1: x $\leftarrow$ x+1
- P2: x $\leftarrow$ x+2
- x = ?

Concurrency and atomicity

Race conditions

Time

Counter = 0	Counter = 0
Reg <- Counter	Reg <- Counter
Reg <- Reg + 1	Reg <- Counter
Counter <- Reg	Reg <- Reg + 1
Reg <- Counter	Reg <- Reg + 1
Reg <- Reg + 1	Counter <- Reg
Counter <- Reg	Counter <- Reg
Counter = 2	Counter = 1

What is the value of x?

Execution-context P1 and P2 run concurrently:

- $x \leftarrow 0$
- P1: $x \leftarrow x+1$
- P2: $x \leftarrow x+2$
- $x = ?$
- $x \leftarrow 0$
- **P1** **P2**
- $\boxed{R \leftarrow x}$ $\boxed{R \leftarrow x}$
- $\boxed{R \leftarrow R+1}$ $\boxed{R \leftarrow R+2}$
- $\boxed{x \leftarrow R}$ $\boxed{x \leftarrow R}$
- x can be 1, 2 or 3 !!!

Safety

Safety = ensuring consistency

- **Mutual exclusion:** shared resources must be updated atomically
- **Condition synchronization:** operations may be delayed if shared resources are in the wrong state, (e.g., read from empty buffer)

Liveness

Liveness = ensuring progress

- No Deadlock: some process can always access a shared resource
- No Starvation: all processes can eventually access shared resources

Expressing Concurrency

- Process creation how do you specify concurrent processes?
- Communication how do processes exchange information?
- Synchronization how do processes maintain consistency?

Concurrent Process Creation

Most concurrent languages offer some variant of the following:

- Co-routines
- Fork and Join
- Cobegin/coend

Co-routines

- Co-routines are only *pseudo-concurrent* and require *explicit transfers of control*
- Co-routines can be used to implement most higher-level concurrent mechanisms.

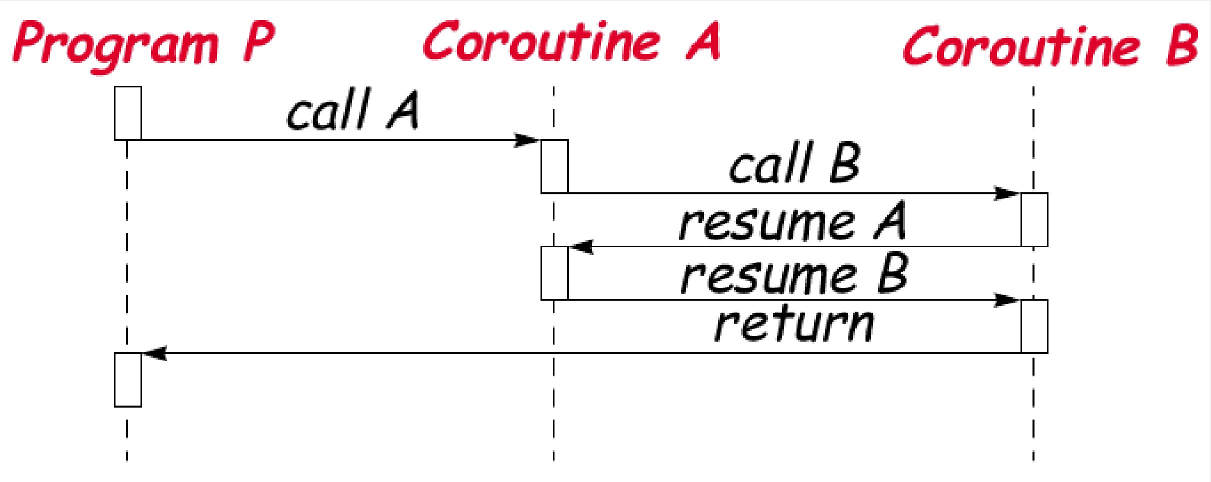


Figure 4

Fork and Join

- Fork can be used to create any number of processes
- Join waits for another process to terminate
- Fork and join are unstructured, so require care and discipline

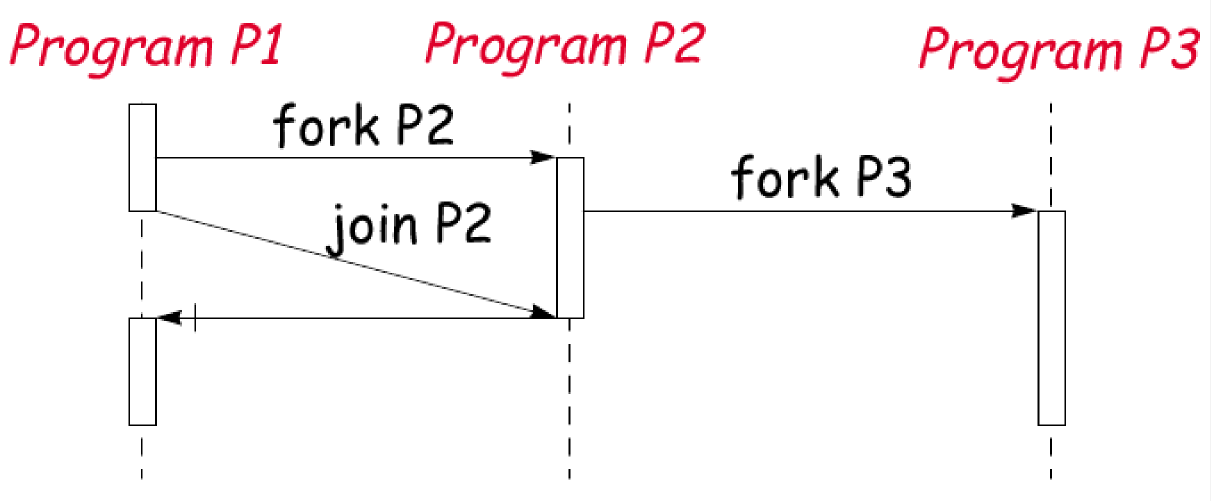


Figure 5

Cobegin/coend

- Cobegin/coend blocks are better structured
- `cobegin S1 | S2 | ... | Sn coend`
- they can only create a fixed number of processes
- The caller continues when all of the coblocks have terminated

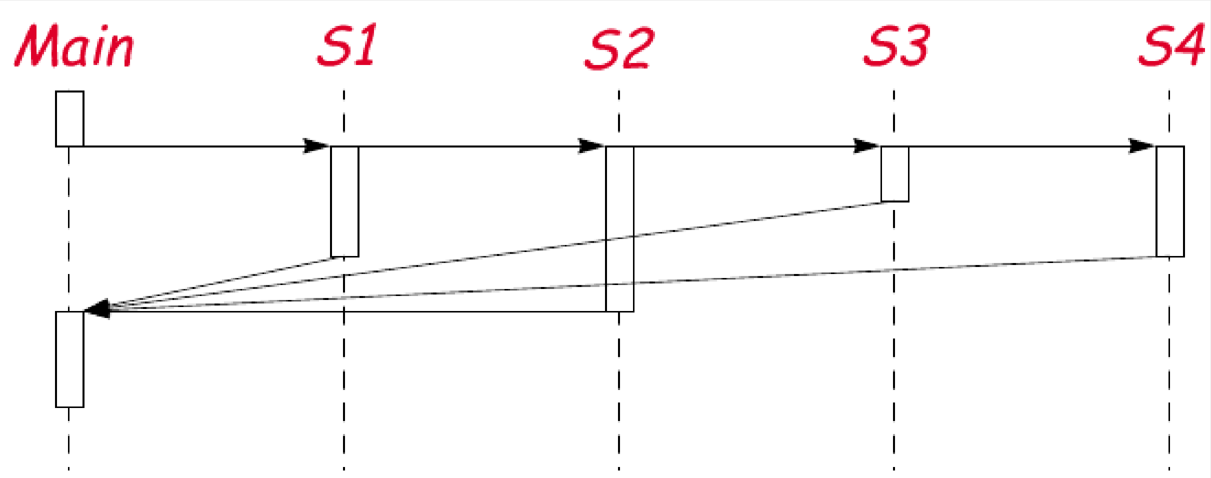
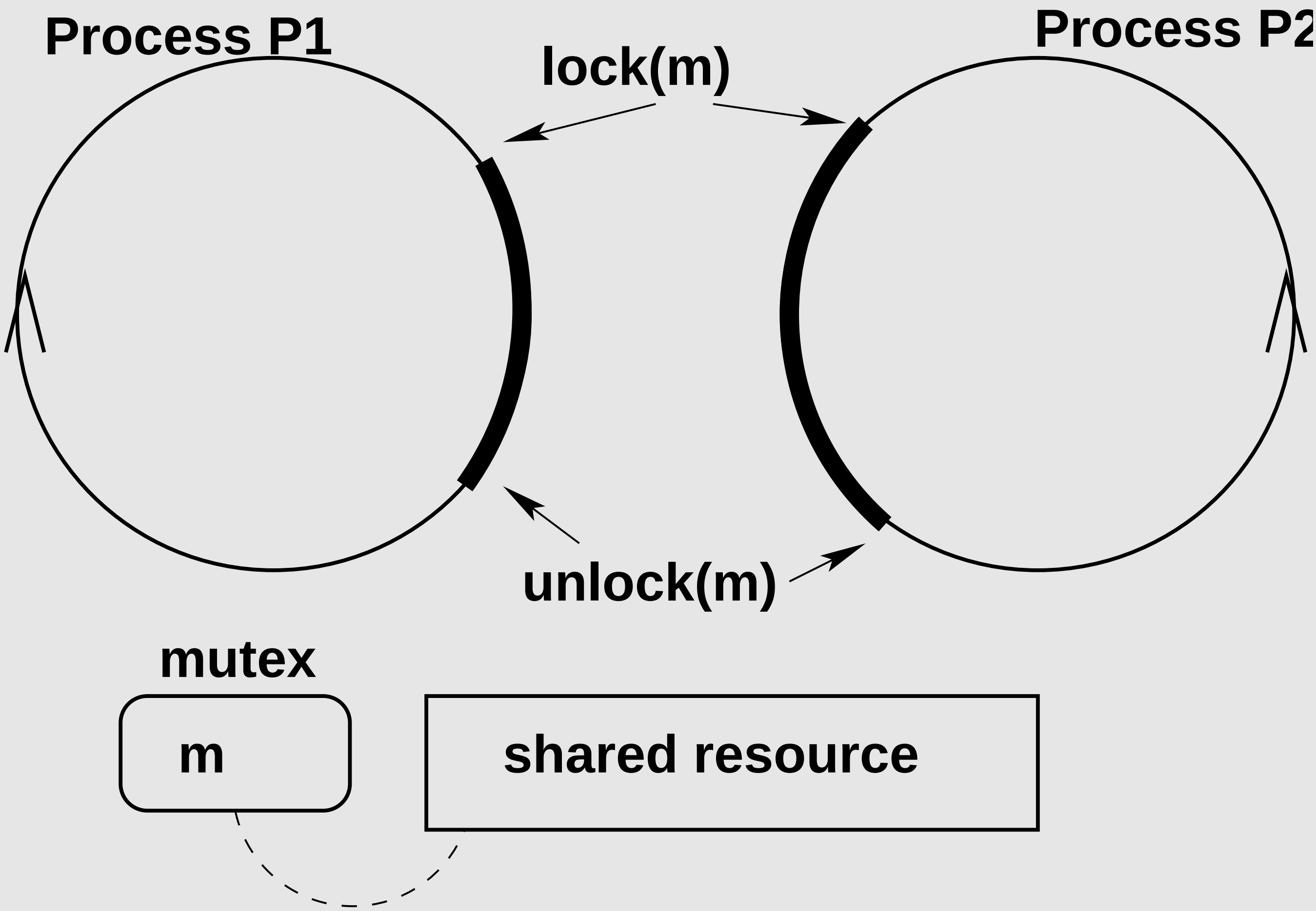


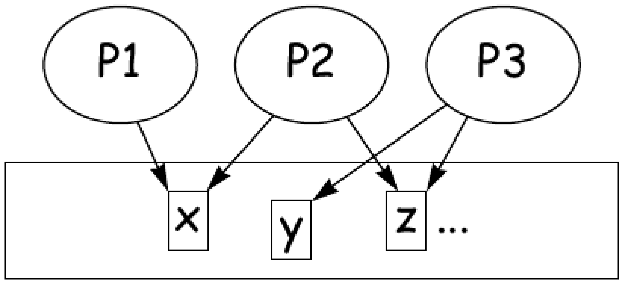
Figure 6

Critical Sections and Mutual Exclusion



Communication and Synchronization

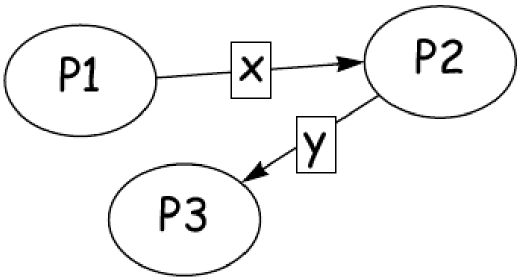
Communication and Synchronization



In approaches based on ***shared variables***, processes communicate *indirectly*. *Explicit synchronization mechanisms* are needed.

In ***message passing*** approaches, *communication and synchronization are combined*.

Communication may be either *synchronous* or *asynchronous*.



Synchronization Techniques

Different approaches are roughly equivalent in expressive power and can be used to implement each other.

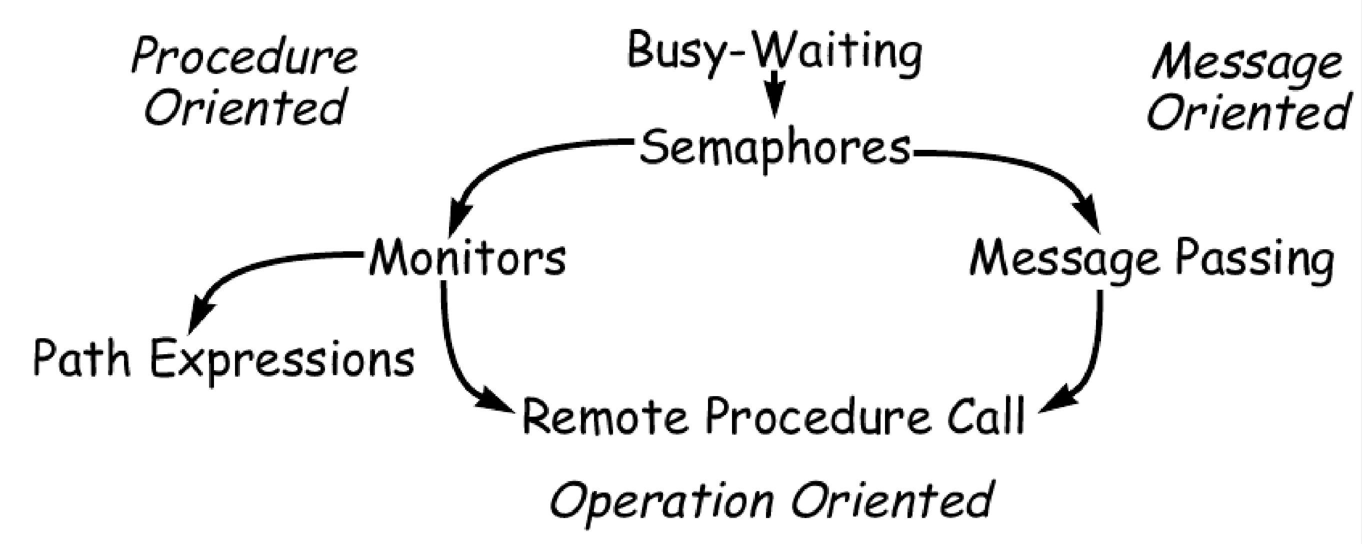


Figure 7

Each approach emphasizes a different style of programming.

Busy-Waiting

- Busy-waiting is primitive but effective
- Processes atomically set and test shared variables
- Condition synchronization is easy to implement
  - to signal a condition, a process sets a shared variable
  - to wait for a condition, a process repeatedly tests the variable
- Mutual exclusion is more difficult to realize correctly and efficiently

Semaphores

- introduced by E.W. Dijkstra (1968)
- a higher-level primitive for process synchronization
- a non-negative, integer-valued variable s with two operations:

– P(s) - delays (waits) until $s > 0$,
  then, atomically executes $s \leftarrow s - 1$

– V(s) - atomically executes $s \leftarrow s + 1$

Programming with semaphores

```
process P1                      process P2
  loop                            loop
    P(mutex)                        P(mutex)
    Critical Section                Critical Section
    V(mutex)                        V(mutex)
    Non-critical Sec                Non-critical Sec
  end                             end
end                             end
```

Monitors

A monitor encapsulates resources and operations that manipulate them:

- operations are invoked like ordinary procedure calls
- invocations are guaranteed to be mutually exclusive
- condition synchronization is realized using wait and signal primitives
- there exist many variations of wait and signal

Other Mechanisms

- Message Passing: combines communication and synchronization
- Remote Procedure Calls (RPC)
- Rendezvous

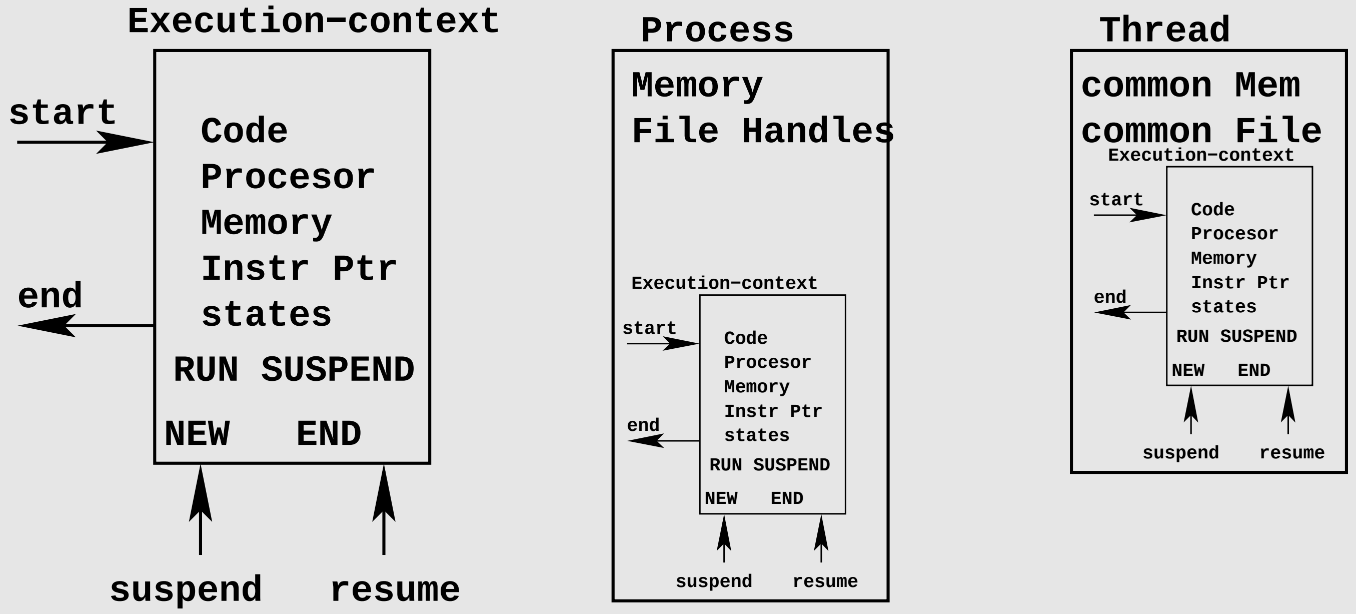
Concurrency and Parallelism

- N processes, N processors – Full Parallelism, Concurrency
- N processes, 1 CPU – no Parallelism, Concurrency
- N processes, M CPU ($M < N$) – some Parallelism,Concurrency
- actual single CPU systems – CPU + DMA (a real processor) + Interrupt-driven I/O (illusion of several processors)
- Virtual Machines – illusion of many CPU's
- concurrency is independent of parallelism
- even with a single CPU, even w/o interrupts (consider Time-sharing systems)

Three basic ideas

- communication: conveying of information from one entity to another
- co-ordination: entities sharing resources are to co-ordinate their activities, for dependable operation
- concurrency: sharing of resurces in the same time frame - CPU, memory, data, code, devices

Execution Contexts



What is a Process?

- *A process is an instance of a running program*, but more precisely -
- runs and provides an environment for a program
- consists of an address space and control point
- it is basic scheduling entity, only one process runs at a time
- contends for and owns various system resources like memory
- requests system services, provided by kernel
- it has a life-time and Life-cycle
- is part of a hierarchy - *init* is ancestor of all

Process context

- User address space: text (code), data, user stack, shared memory, etc.
- Control information - proc structure, kernel stack, address translation tables
- Credentials - user and group ID's
- Environmental variables -
  - a set of strings of the form "var = value"
  - library provides functions to manipulate these
  - inherited from the parent process
  - while invoking a new process, option of new environment
- Hardware context - IP(PC), SP, registers, PSW, MMU regs, FPU regs

Threads

- Consider a file server that has several "Threads of Control"
- The server will block occasionally for disk I/O
- While one thread is sleeping, another could continue its execution
- Net result: higher throughput, better performance

Threads - 2

- sometimes called Light Weight Process (LWP)
- they are like mini-processes
- runs strictly sequentially, own PC, stack
- share CPU in time-share manner
- only on multiple CPU (multiprocessor) can they really in parallel
- can create child threads, block on system calls
- while one thread is blocked, other in the same process can run
- Creation of a Thread is 10 to 25 times faster than a Processes, e.g. time for 50,000 creations:

Platform	fork()			pthread_create()		
2.4GHz Xeon	real	user	sys	real	user	sys
(2 cpus/node)	54.9	1.5	20.8	1.6	0.7	0.9

Per thread items vs. per process items

Per thread items:

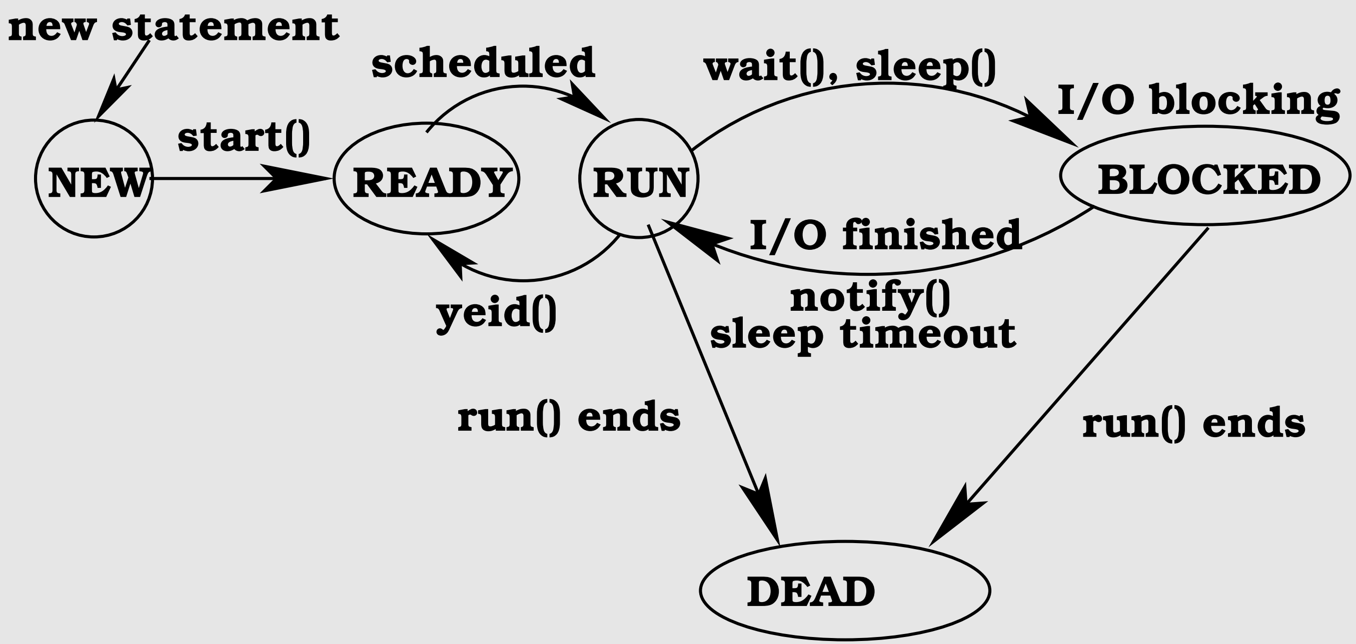
- Program counter
- stack, SP
- Register set (How?)
- Child threads
- State

Per process items:

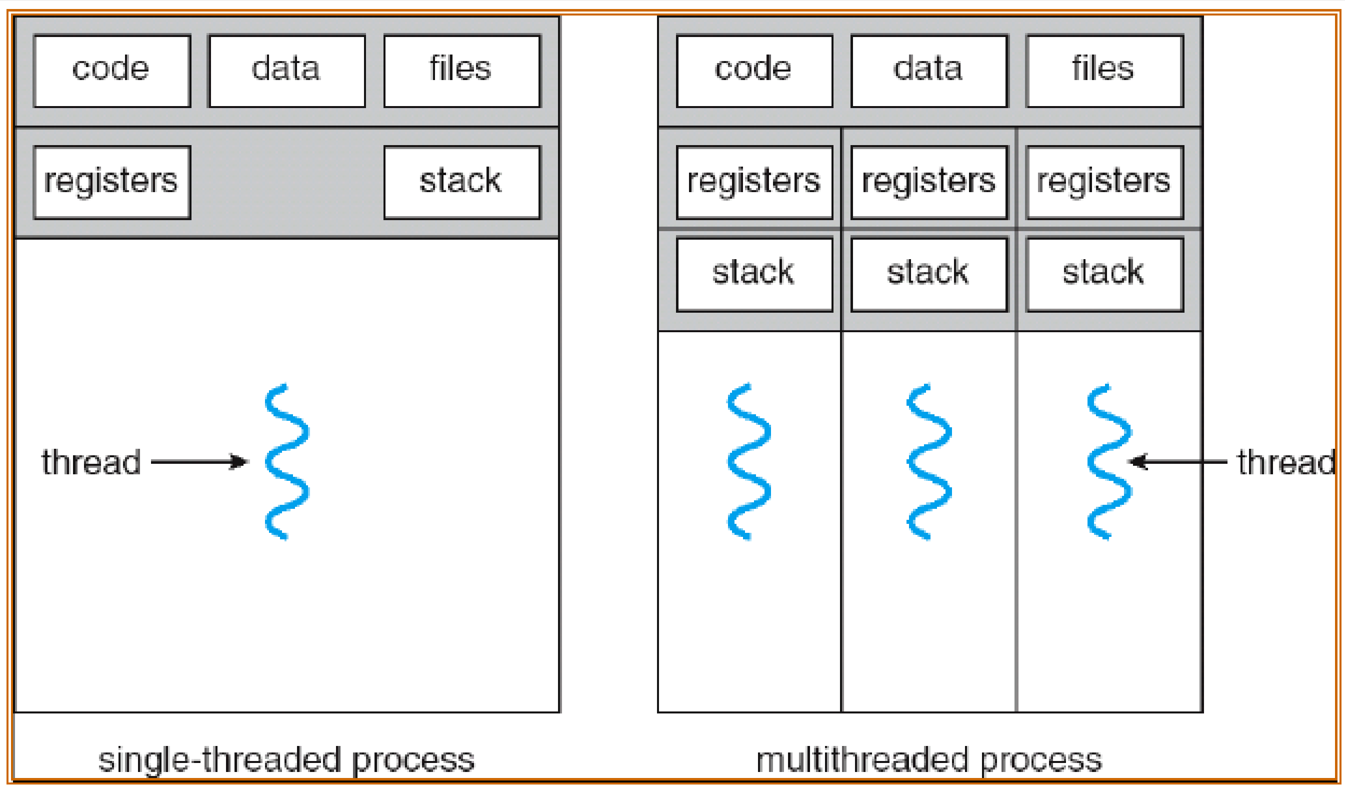
- Address space, global variables;
- Open files, Timers, Signals;
- Child processes

- Semaphores
- Accounting information

Life-cycle of a Thread



Single and Multithreaded Processes



Single and Multithreaded Processes

- Threads encapsulate concurrency: Active component
- Address spaces encapsulate protection: Passive part
  - Keeps buggy program from trashing the system
- Why have multiple threads per address space?

On writing concurrent code

- High-level constructs desirable
- use low-level constructs like - threads, semaphores, mutex, condition variables
- parallel begin: `initiate`
- shared variables: `shared int i`
- Critical Region: `region i do { ...}`
- Conditional Critical Region: delays a process until components of a shared variable v satisfy a condition B, `await()`
- Semaphore: - define, init and `Wait()` and `Signal()`
- we use a pre-processor to map these high-level constructs to functions available in C library