Computer Architecture

Process Management

Process Management

- Processes
- Threads
- Interprocess Communication (IPC)
- CPU Scheduling
- Process Synchronization
- Deadlocks

Processes

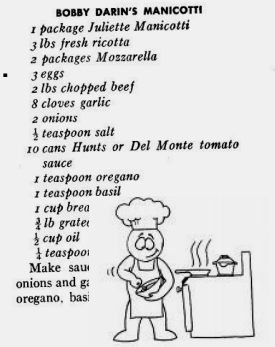
A *process* is a set of identifiable, repeatable actions which are ordered in some way and contribute to the fulfillment of an objective.

(General definition)

A *process* is a program in execution.

(Computer oriented definition)

- Program: static, passive
  A cooking recipe is a program.
- Process: dynamic, active
  Acting according to the recipe (cooking) is a process.



Process Model

- Several processes are working *quasi*-parallel.
- A *process* is a unit of work.
- Conceptually, each process has its own virtual CPU.
  In reality, the real CPU switches back and forth from process to process.

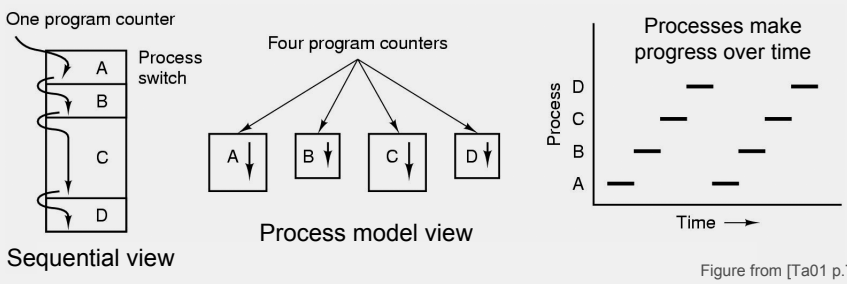


Figure from [Ta01 p.72]

Processes

A process may be described as either (more or less)

a) CPU-bound
   spends more time doing computations – few very long CPU bursts.

b) I/O-bound
   spends more time doing I/O than computations – many short CPU bursts.

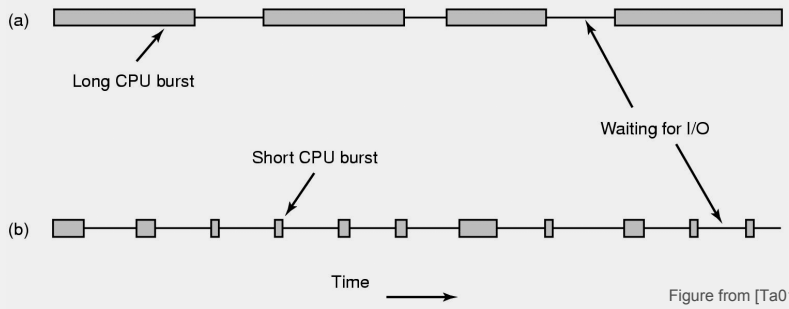
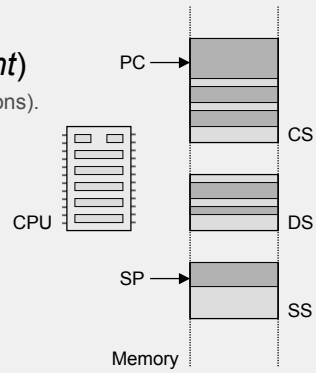


Figure from [Ta01 p.134]

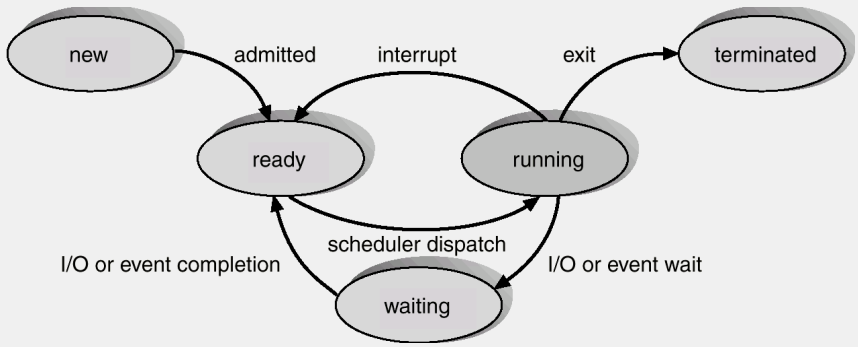
Address Space

A *process* is an executing program, and encompasses the current values of the program counter, of the registers, of the variables and of the stack.

- code section (text section or *segment*)
  This is the actual program code (the machine instructions).
- data section (data segment)
  This segment contains global variables (global to the process, not global to the computer system).
- stack section (stack segment)
  The stack contains temporary data (local variables, return addresses, function parameters).



Process States

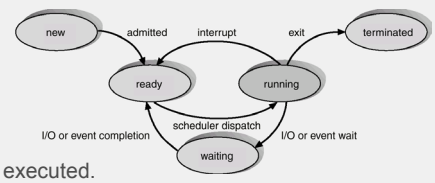


Note: Only in the *running* state the process needs CPU cycles, in all other states it is actually ‚frozen' (or nonexistent any more).

Figure from [Sil00 p.89]

Process States

- **New**
  The process is created. Resources are allocated.
- **Ready**
  The process is ready to be (re)scheduled.
- **Running**
  The CPU is allocated to the process, that is, the program instructions are being executed.



- **Waiting**
  The process is waiting for some event to occur. Without this event the process cannot continue – even if the CPU would be free.
- **Terminated**
  Work is done. The process is taken off the system (off the queues) and its resources are freed.

Processes

Events at which processes are created

- Operating System Start-Up
  Most of the system processes are created here. A large portion of them are background processes (daemons).
- Interactive User Request
  A user requests an application to start.
- Batch job
  Jobs that are scheduled to be carried out when the system has available the resources (e.g. calendar-driven events, low priority jobs)
- Existing process gives birth
  An existing process (e.g. a user application or a system process) creates a process to carry out some related (sub)tasks.

Process Creation

- Parent process creates a child process
  which in turn may create other processes, forming a tree of processes.
- Resource sharing
  - Parent and child share all resources.
  - Child shares subset of parent's resources.
  - Parent and child share no resources.
- Execution
  - Parent and child execute concurrently.
  - Parent waits until child terminates.
- Address Space
  - Child is copy of parent
  - Child has program loaded into it

System calls for creating a child process:
Unix: `fork()`
Windows: `CreateProcess()`

`fork()` example


```
#include <stdio.h>

void main()
{
   int result;
   printf(„Parent, my pid is: %d\n", getpid());
   result = fork();
   if (result == 0) { /* child only */
      printf(„Child, my pid is: %d\n", getpid());
      ...
   } else { /* parent only */
      printf(„Parent, my pid is: %d\n", getpid());
      ...
   }
}
```

system call that tells a process its `pid` (process identifier) which is a unique process number within the system.

Executed by child

Executed by parent

from here on think *parallel*

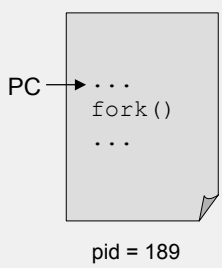
`fork()` example

Terminal output:

```
Parent, my pid is: 189
Child, my pid is: 190
Parent, my pid is: 189
```

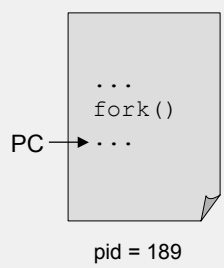
order depends on whether parent or child is scheduled first after `fork()`.

Before **`fork()`**

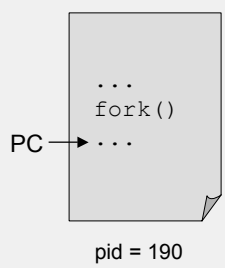


pid = 189

After **`fork()`**

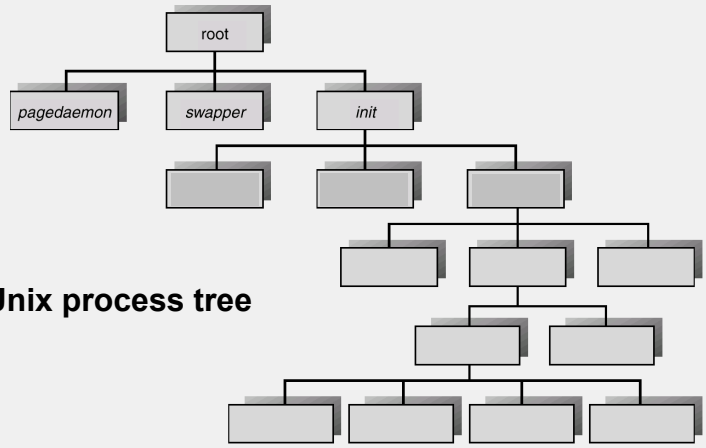


pid = 189



pid = 190

Process Creation



A Unix process tree

Figure from [Sil00 p.96]

Process Termination

Events at which processes are terminated

- Process asks the OS to delete it
  Work is done. Resources are deallocated (memory is freed, open files are closed, used I/O buffers are flushed).
- Parent terminates child
  - Child may have exceeded allocated resources.
  - Task assigned to child is no longer required.
  - Parent‘s parent is exiting.
    Some OS do not allow a child to continue when its parent terminates. Cascading termination (a sub tree is deleted).

System calls for self-termination of a process:
Unix: `exit()`
Windows: `ExitProcess()`

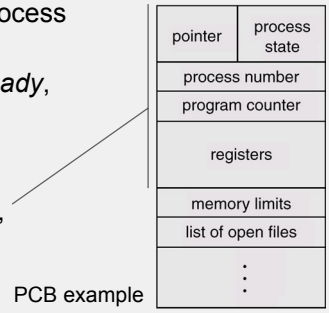
Process Control Block

- Operating system maintains a process table
- Each entry represents a process
- Entry often termed *PCB* (process control block)

A PCB contains all information about a process that must be saved when the process is switched from *running* into *waiting* or *ready*, such that it can later be restarted as if it had never been stopped.

- Info regarding process management,
- regarding memory occupation
- and open files.

Figure from [Sil00 p.89]



PCB example

Process Control Block

Process management	**Memory management**	**File management**
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	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		

Typical fields of a PCB

Table from [Ta01 p.80]

Context Switch

The task of switching the CPU from one process to another is termed *context switch* (sometimes also *process switch*):

- Saving the state of old process
  Saving the current *context* of the process in its PCB.
- Loading the state of new process
  Restoring the former context of the process from its PCB.

Context switching is pure administrative overhead. The duration of a switch lies in the range of 1 ... 1000 µs. The switch time depends on the hardware. Processors with multiple sets of registers are faster in switching. Context switching poses a certain bottleneck, which is one reason for the introduction of *threads*.

Context Switch

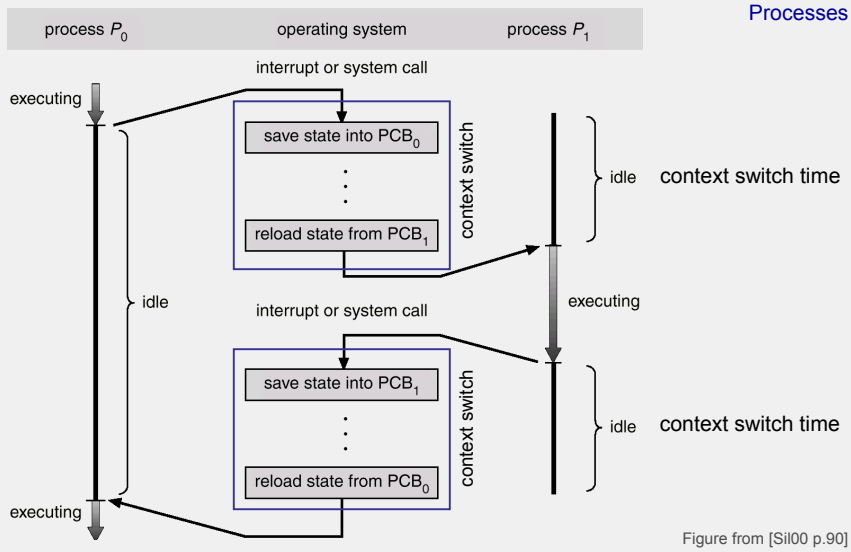


Figure from [Sil00 p.90]

Scheduling

On a uniprocessor system there is only one process running, all others have to wait until they are scheduled. They are waiting in some **scheduling queue**:

- Job Queue
  Holds the future processes of the system.
- Ready Queue (also called CPU queue)
  Holds all processes that reside in memory and are ready to execute.
- Device Queue (also called I/O queue)
  Each device has a queue holding the processes waiting for I/O completion.
- IPC Queue
  Holds the processes that wait for some IPC (inter process communication) event to occur.