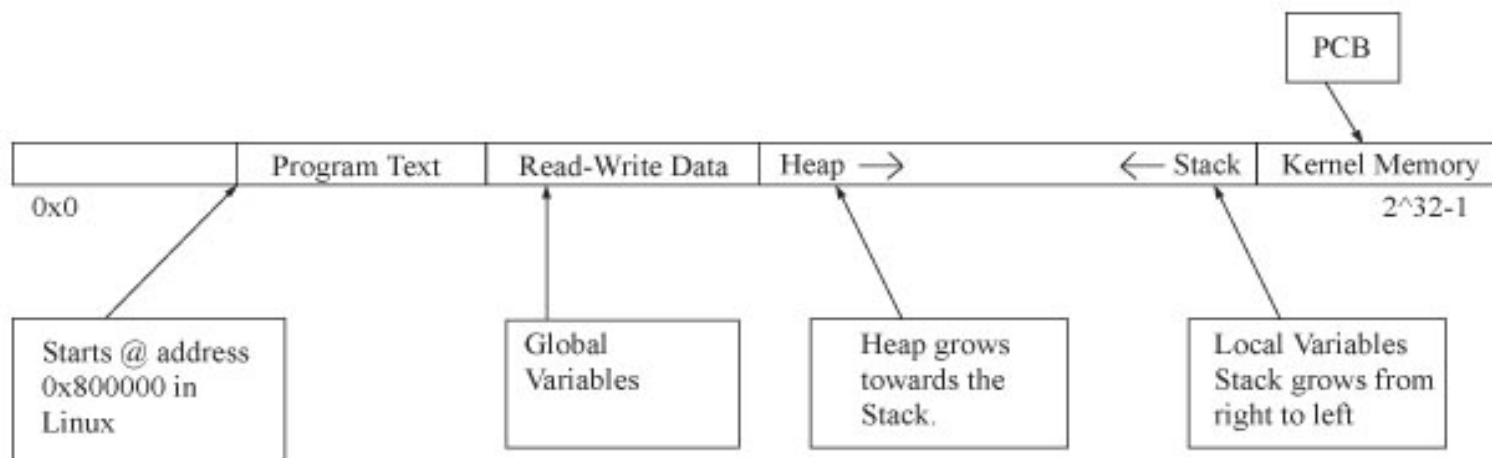


The Basic Concepts: Processes and Threads

A Program

- What are the key parts?
 - The instructions (code)
 - Stack
 - Heap
 - Program counter

Memory Layout for a Linux Program



Taken from
<http://read.cs.ucla.edu/111/2006spring/notes/lec3>

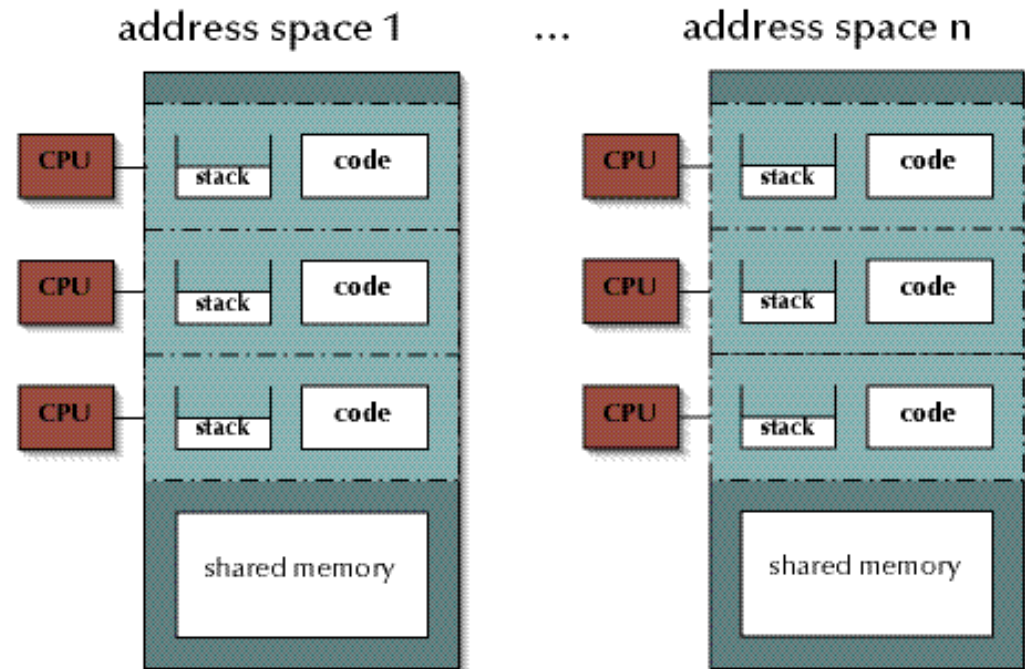
A Linux Program: Addresses

- Addresses
 - Program: where your main begins (0x8000000)
 - Data: where the global variables are created
 - Heap: where your dynamically allocated memory is located (-0x8660000)
 - Stack: begins where you push your first local variable

```
#include<stdio.h>
#include<stdlib.h>
int x; //create a global variable
int main (int c, char **v)
{
    void *p = malloc (1); //dynamic alloc memory
    printf ("Prog %p\n Data %p\n //print addresses for Prog, Data,
           Heap %p\n Stack %p\n", //Heap and Stack
           &main, &x, p, &c);
    return 1;
}
```

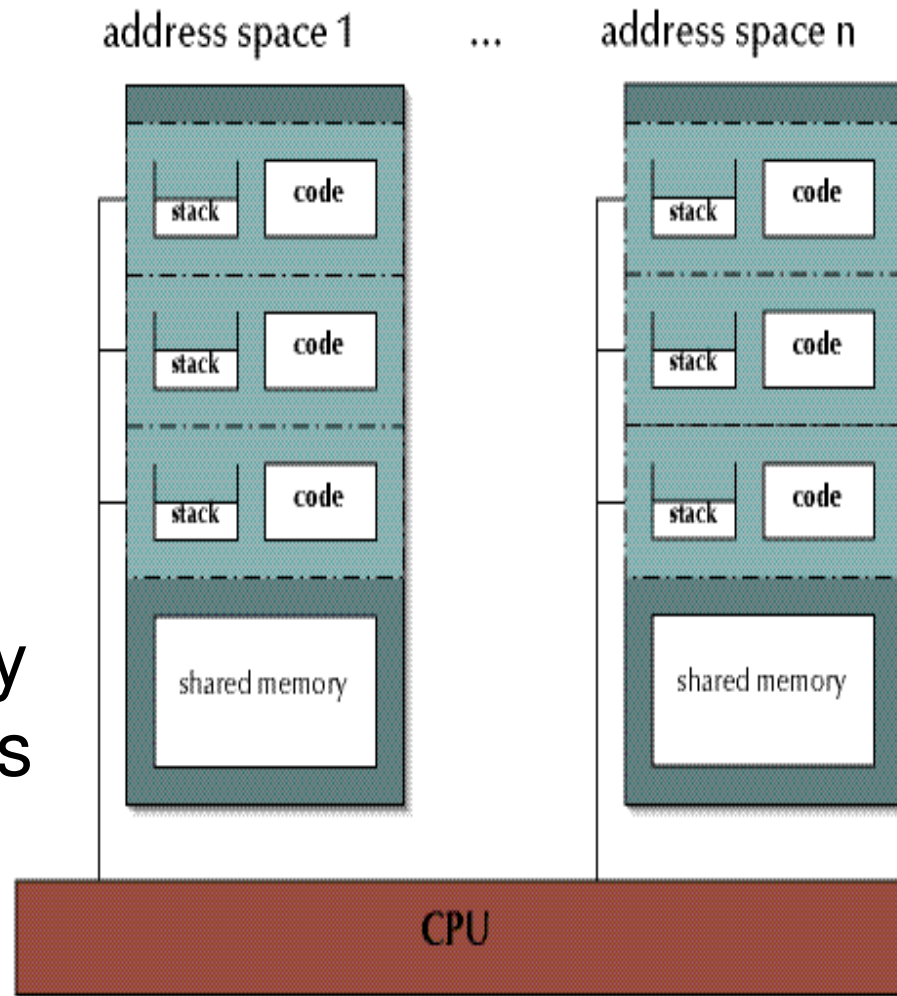
1-CPU per Control Flow

- For specific configurations only:
 - distributed μ controllers
 - physical process control systems: 1 cpu per task connected via a typ. fast bus-system (VME, PCI)
- No need for process management



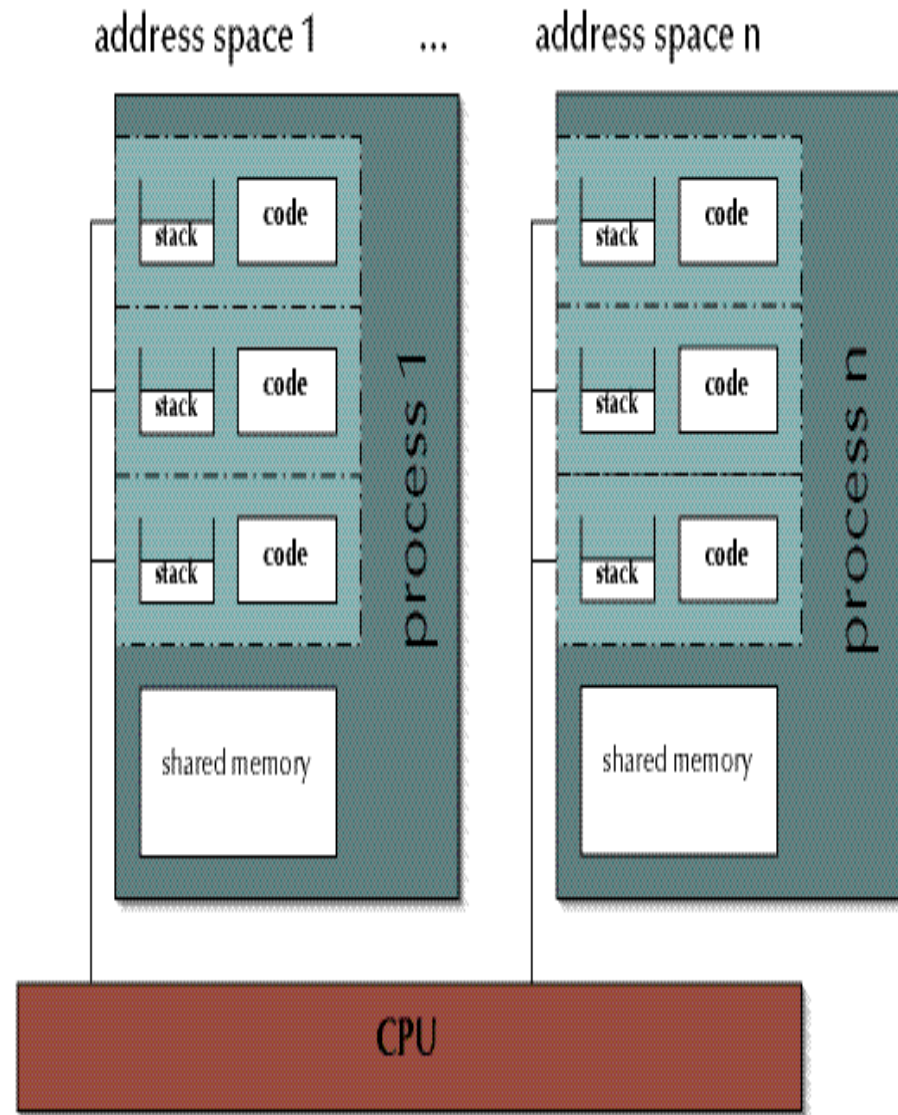
1 CPU for all control-flows

- OS: emulate one CPU for every control-flow
 - **multi-tasking** operating system
- support for memory protection becomes essential



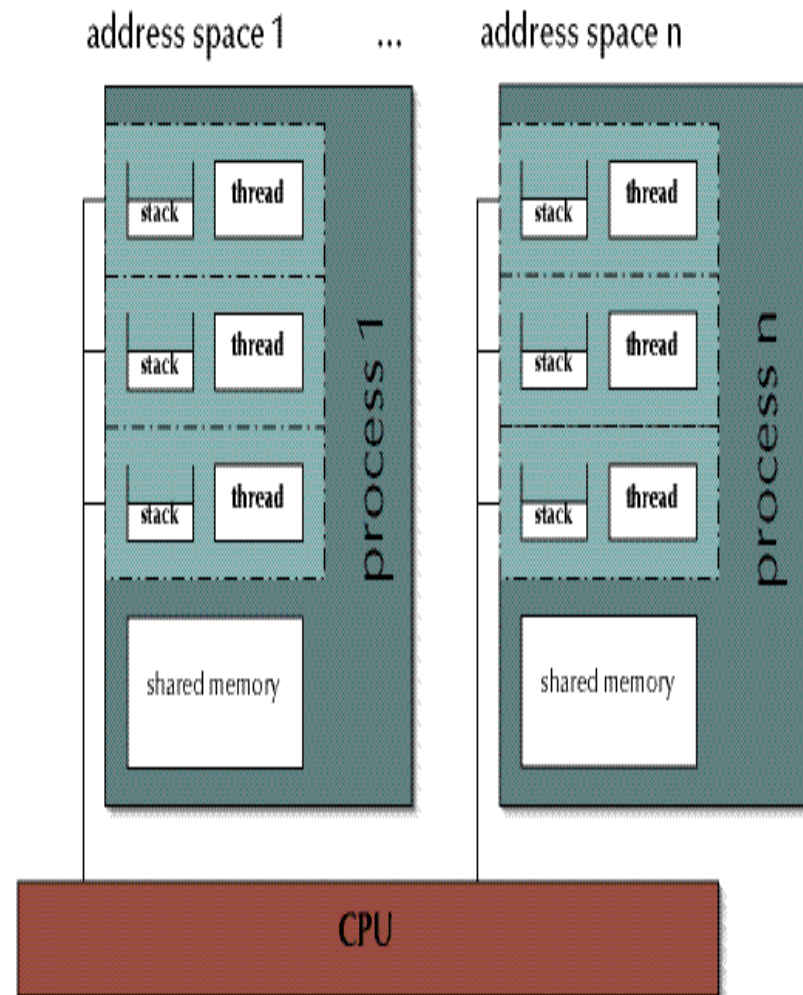
Processes

- **Process ::=**
address space +
control flow(s)
- Kernel has full
knowledge about all
processes as well as
their *requirements*
and current *resource*.
(see below)



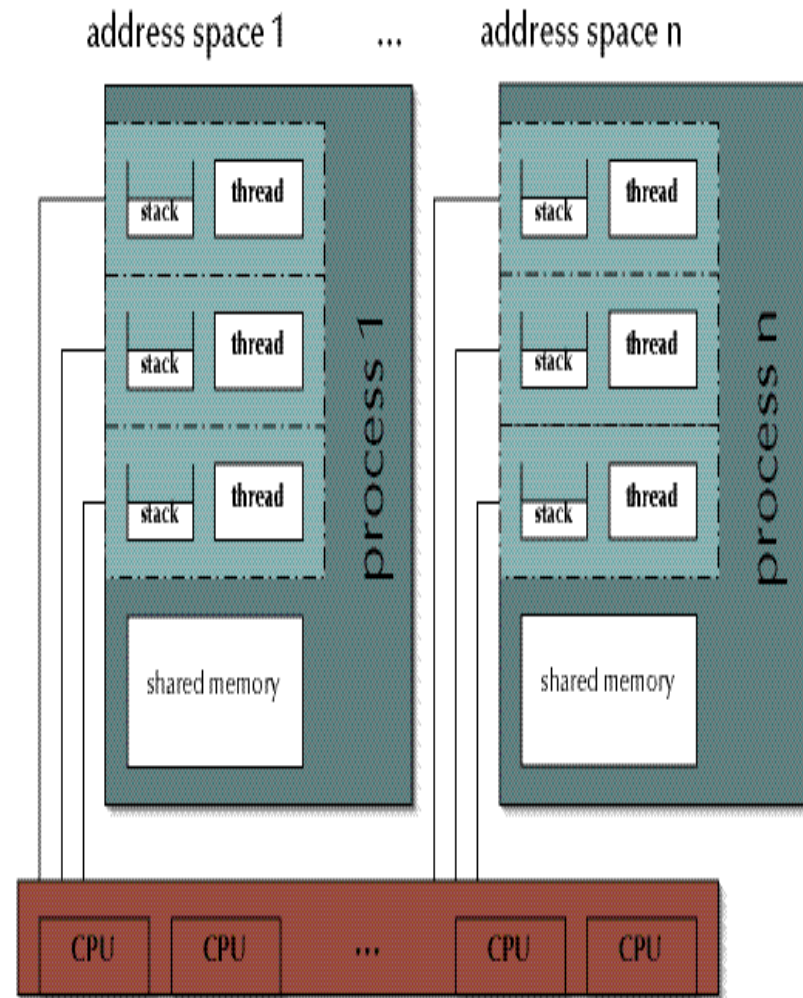
Threads

- **Threads** (individual control-flows) can be handled:
- *Inside the kernel:*
 - kernel scheduling
 - I/O block-releases according to external signal
- *Outside the kernel:*
 - user-level scheduling
 - no signals to threads



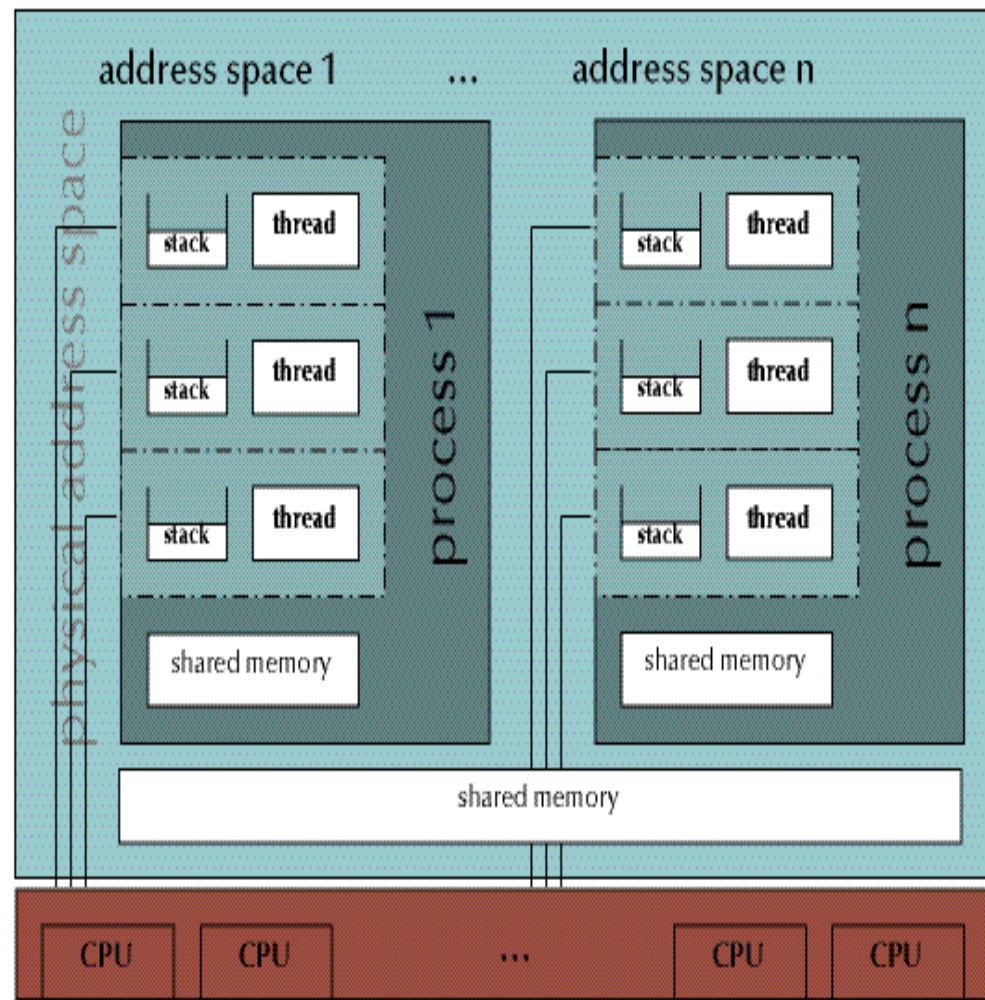
Multi-Processor Systems

- The kernel may execute multiple processes at a time.
 - Address space and resource restrictions of individual CPUs and processes/threads need to be considered.
 - Caching, synchronization, and memory protection need to be adapted.



Symmetric Multi-Processing (SMP)

- All CPUs share the same physical address space (and access to resources)
 - processes/threads can be executed on any available CPU



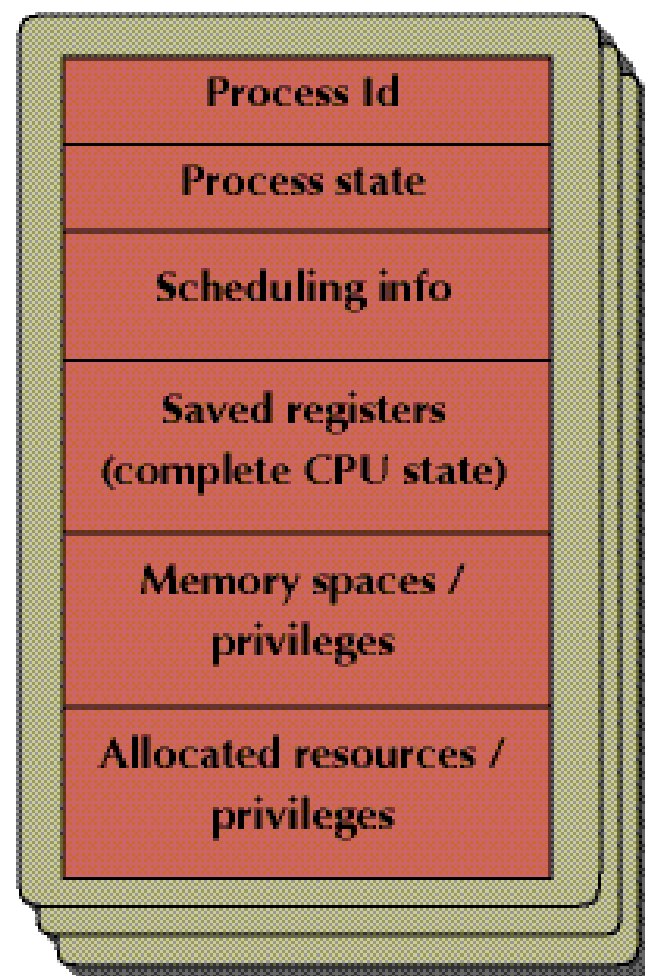
Processes and Threads

- Process or Thread?
 - Processes can share memory
 - the exact interpretation of threads is different in different operating systems:
- Threads can be regarded as a group of processes, which share some resources (+ process-hierarchy)
- Due to the overlap in resources, the attributes attached to threads are less than for ‘first-class-citizen-processes’
- Thread switching and inter-thread communications can be more efficient than on full-process-level
- Scheduling of threads depends on the actual thread implementations:
 - e.g. user-level control-flows, which the kernel has no knowledge about at all
 - e.g. kernel-level control-flows, which are handled as processes with some restrictions

Process Control Blocks

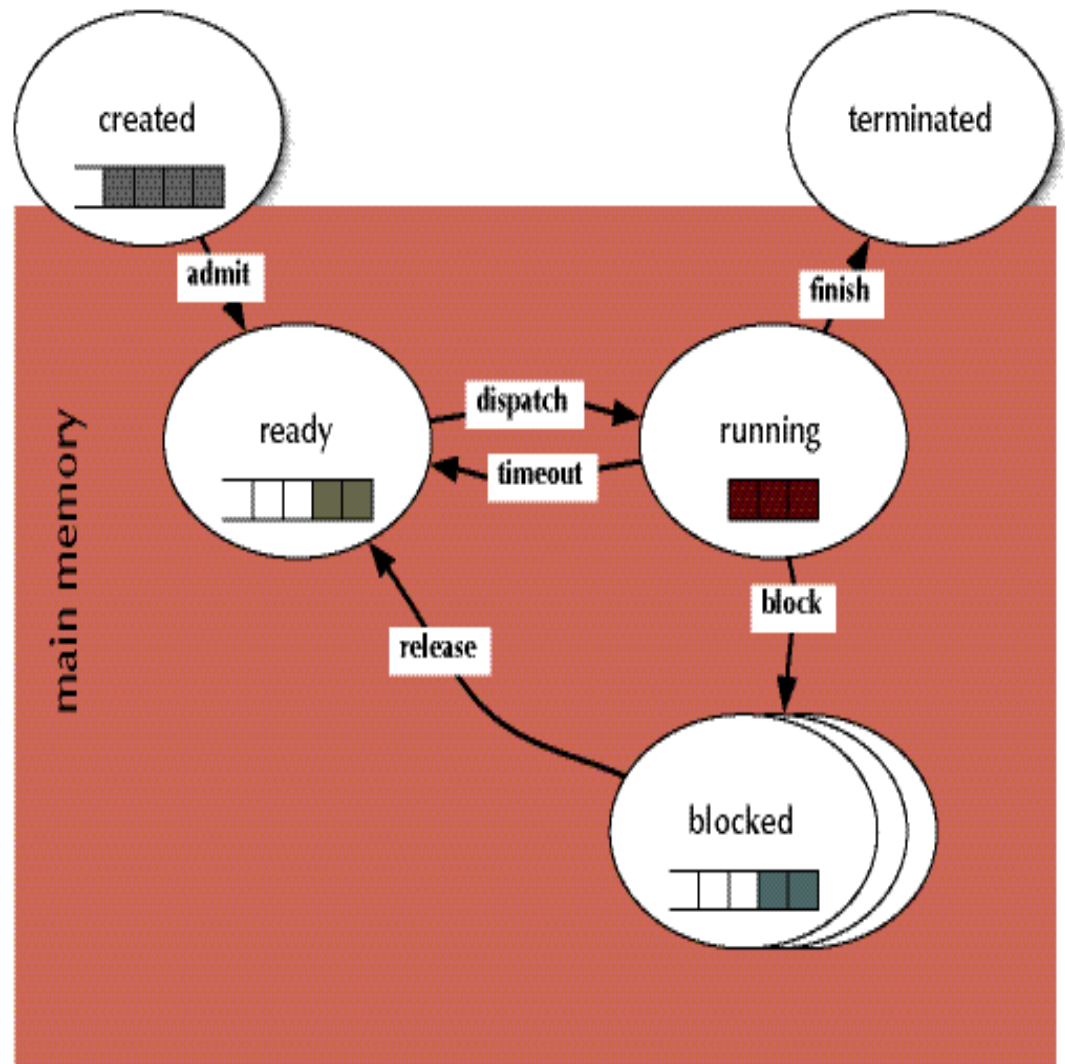
- **Process Id**
- **Process state:**
{created, ready, executing, locked, suspended, ...}
- **Scheduling info:**
priorities, deadlines, consumed CPU-time, ...
- **CPU state:**
saved/restored information while context switches (incl. the program counter, stack pointer, ...)
- **Memory spaces / privileges:**
memory base, limits, shared areas, ...
- **Allocated resources / privileges:**
open and requested devices and files
- PCBs are usually enqueued at a certain state or condition

Process Control Blocks (PCBs)



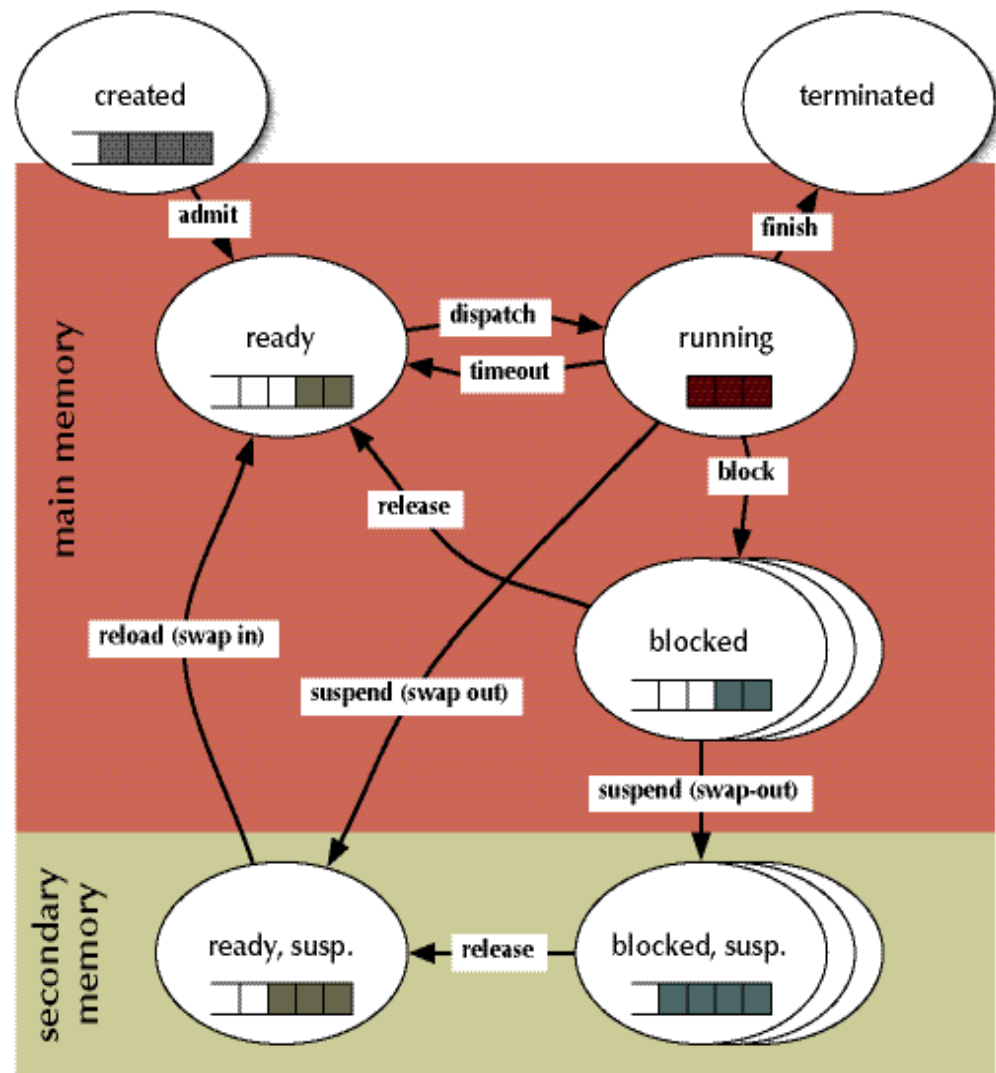
Process States

- **created**: the task is ready to run, but not yet considered by any dispatcher
 - waiting for admission
- **ready**: ready to run
 - waiting for a free CPU
- **running**: holds a CPU and executes
- **blocked**: not ready to run
 - waiting for a resource to become available



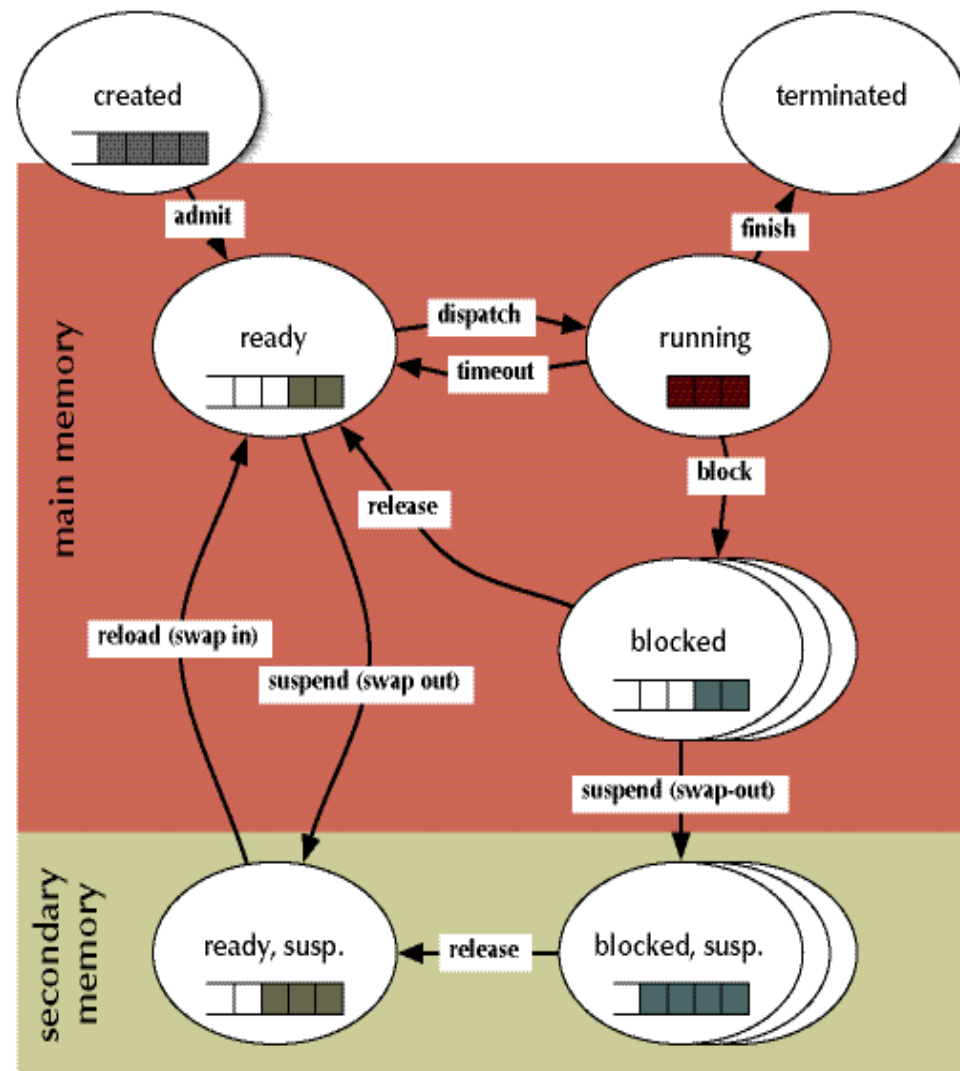
Process States: Suspended

- **created**: the task is ready to run, but not yet considered by any dispatcher
 - waiting for admission
- **ready**: ready to run
 - waiting for a free CPU
- **running**: holds a CPU and executes
- **blocked**: not ready to run
 - waiting for a resource
- **suspended states**: swapped out of main memory (not time critical processes)
 - waiting for main memory space (and other resources)

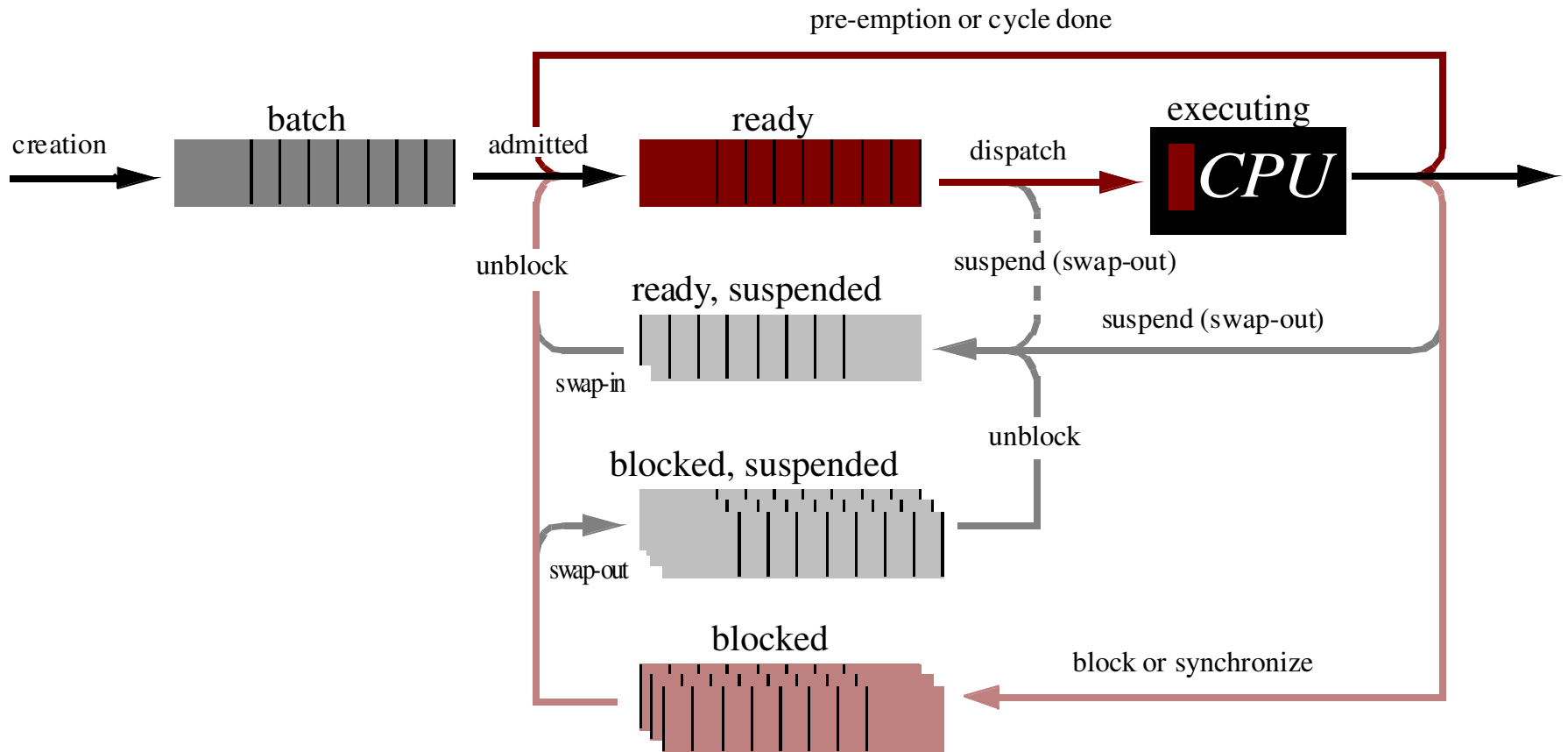


Process States: Suspended

- **created**: the task is ready to run, but not yet considered by any dispatcher
 - waiting for admission
- **ready**: ready to run
 - waiting for a free CPU
- **running**: holds a CPU and executes
- **blocked**: not ready to run
 - waiting for a resource
- **suspended states**: swapped out of main memory (not time critical processes)
 - waiting for main memory space (and other resources)
- **Dispatching and suspending can be independent modules here**



Process States



Unix Processes

```
pid = fork ();
```

- Results in a *duplication* of the *current* process
 - returning **0** to the newly created process (the 'child' process)
 - returning the **process id** of the child process to the creating process (the 'parent' process) or **-1** for a failure
- Frequent usage:

```
if (fork () == 0) {  
    ... the child's task ... often  
    ... exec ("abs path to executable file", "args");  
    exit (0);      /* terminate child process */  
} else {  
    ... the parent's task ... waits for child  
    pid = wait ();  
}
```

Unix Pipes

```
int data_pipe [2], c, rc;

if (pipe (data_pipe)==-1){
    perror ("no pipe");exit(1);
}

if (fork () == 0) {
    close (data_pipe [1]);
    while ((rc = read
        (data_pipe [0],&c,1))>0){
        putchar (c);
    }
    if (rc == -1) {
        perror ("pipe broken");
        close (data_pipe [0]);
        exit (1);
    }
    close (data_pipe [0]);
    exit (0);
} else {
    close (data_pipe [0]);
    while ((c = getchar ()) > 0) {
        if (write
            (data_pipe[1],&c,1)==-1){
            perror ("pipe broken");
            close (data_pipe [1]);
            exit (1);
        };
    }
    close (data_pipe [1]);
    pid = wait ();
}
```

Concurrent Programming

- Requirement
 - Concept of **tasks**, **threads** or other **potentially concurrent entities**
- Frequently requested
 - Support for **management** or concurrent entities (create, terminate, ...)
 - Support for **contention management** (mutual exclusion, ...)
 - Support for **synchronization** (semaphores, monitors, ...)
 - Support for **communication** (message passing, shared memory, rpc, ...)
 - Support for **protection** (tasks, memory, devices, ...)

Language Candidates

- Ada95, Chill, Erlang
- Occam, CSP
- Java, C#
- Modula-2
- Lisp, Haskell, Caml, Miranda
- Smalltalk, Squeak
- Prolog
- Esterel, Signal
- Without any support for concurrency: Eiffel, C, C++, Pascal, Fortran, Cobol, Basic...

C-libraries & interfaces

- POSIX: Portable Operating System Interface
- MPI (message passing interface)

Implicit Concurrency and Functional Languages

```
qsort [] = []
```

```
qsort (x:xs) = qsort [y | y <- xs, y < x] ++ [x] ++ qsort [y | y <- xs, y >= x]
```

- Quicksort in two lines!
- Line 1:
 - qsort of empty array is empty array
- Line 2:
 - qsort of a list containing one element and the rest is the quick sort of all elements out of the remainder that are smaller than x concatenated with x concatenated with all elements in the remainder that are larger than x
- Strict functional programming is **side-effect free**
 - Parameters can be evaluated independently + concurrently

Lazy Evaluation

- Some functional languages allow for '**lazy evaluation**', i.e. sub-expressions are not necessarily evaluated completely:

```
borderline = (n /= 0) && (g (n) > h (n))
```

- if n equals zero the evaluation of g(n) and h(n) can be stopped (or not even be started)
- concurrent program parts need to be interruptible in this case

- (Lazy) sub-expression evaluations in imperative languages still assume sequential execution: eg in Ada

```
if Pointer /= nil and then Pointer.next =  
    nil then ...
```

Ada95

- We will now briefly discuss Chapters 1 & 2 in Ben-Ari, before going on to talk about Ada95