Australian National University

C03

Processes and Threads

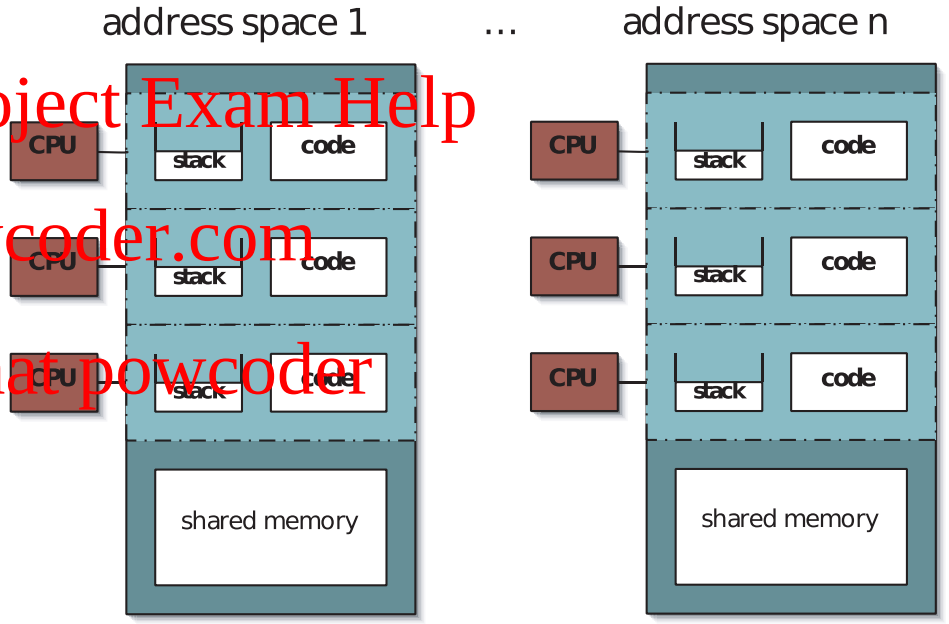
Operating system support

Process states

Unix processes

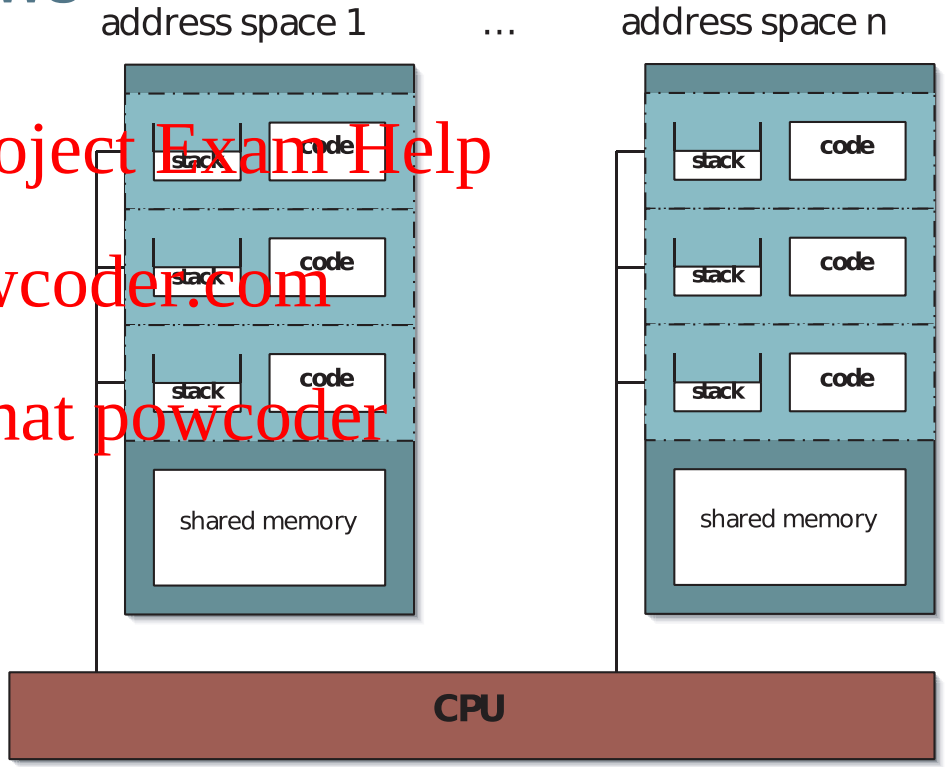
One CPU per control flow

- Specialized configurations e.g.:
  - Distributed µcontrollers
  - Physical process control systems: 1 CPU per task, connected via a bus-system
- Process management (scheduling) not required
- Must coordinate access to shared memory



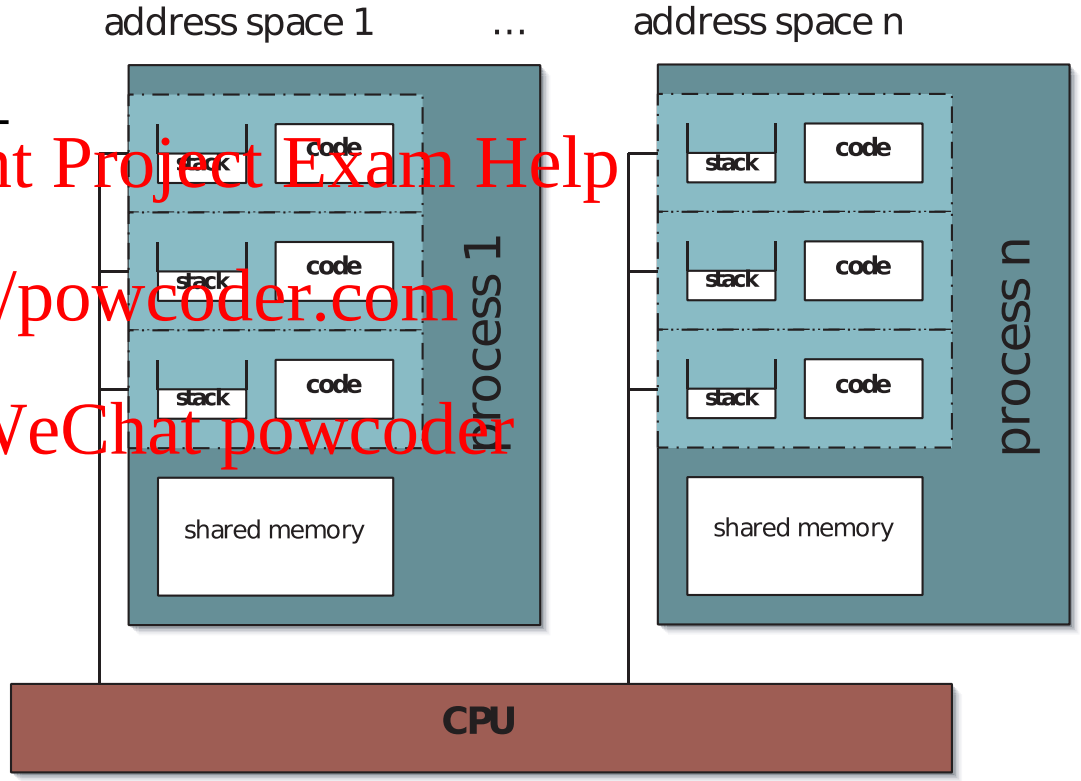
One CPU for all control flows

- OS: emulate one CPU for every control flow:
  Multi-tasking operating system
  stack
  - Process management (scheduling) required
  - Must coordinate access to shared memory
  - Support for memory protection essential



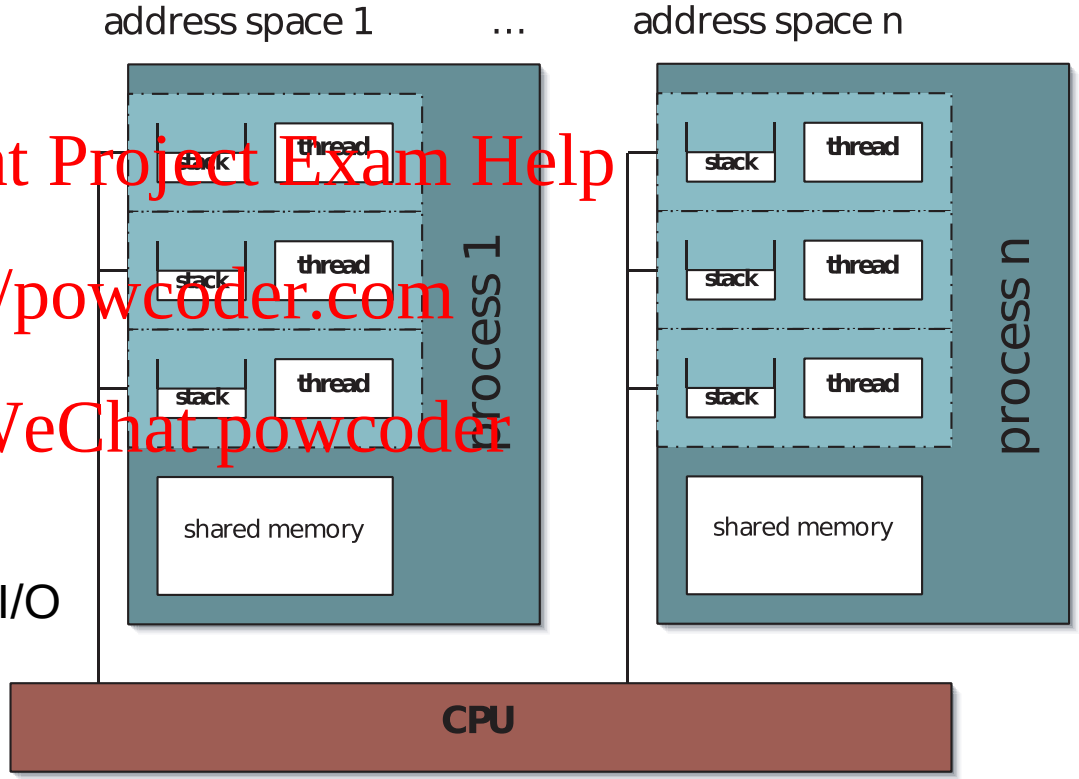
Processes

- Process ::= address space + control flow(s)
- Kernel has full knowledge about all processes
  - state
  - requirements
  - currently held resources



Threads

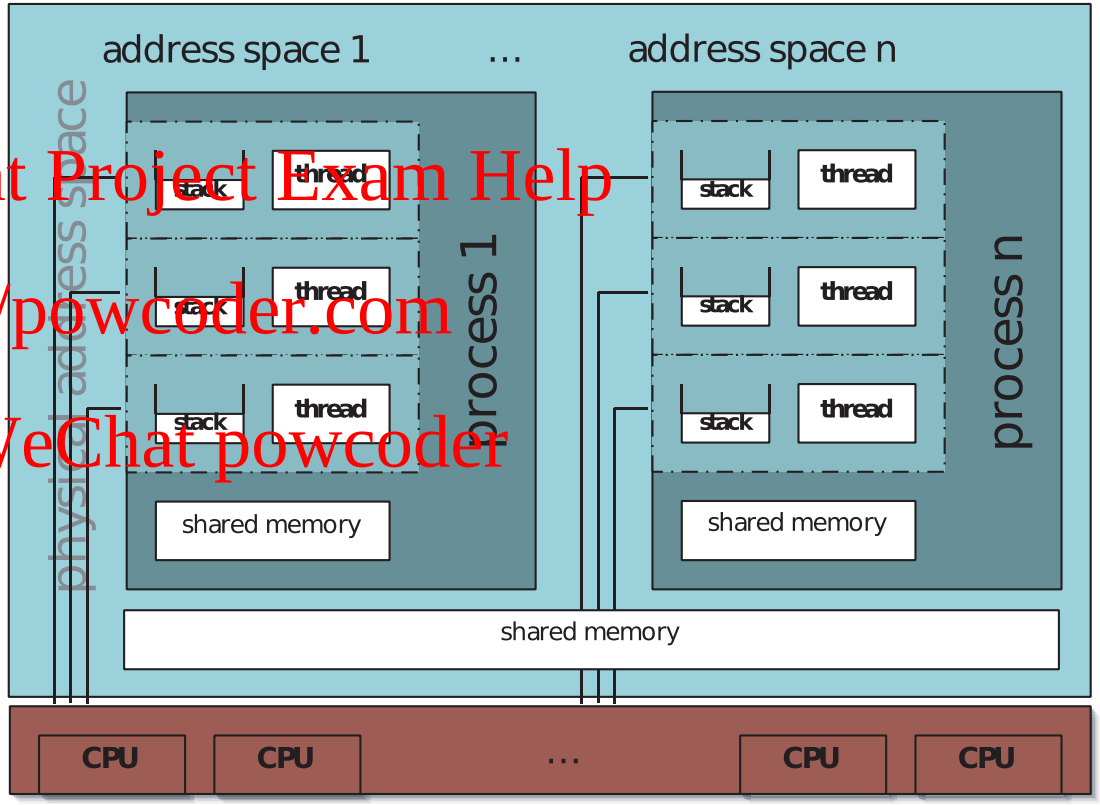
- Threads (individual control flows) can be handled:
  - Inside the OS:
    - Kernel scheduling
    - Thread performs I/O
  - Outside the OS:
    - User-level scheduling
    - Parent process performs I/O



SMP

Symmetric Multi-Processing

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



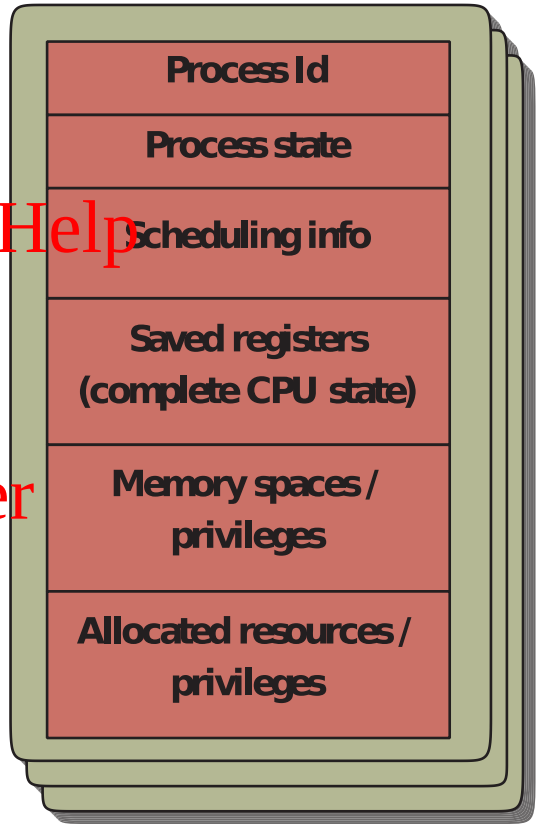
Processes or Threads ?

Specific definition of threads depends on operating system and context:

- Threads ≡ 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 communication can be more efficient than switching on process level
- Scheduling of threads depends on the actual thread implementations:
  - user-level control-flows (no kernel support)
  - kernel-level control-flows, handled as processes with some restrictions

Process Control Blocks

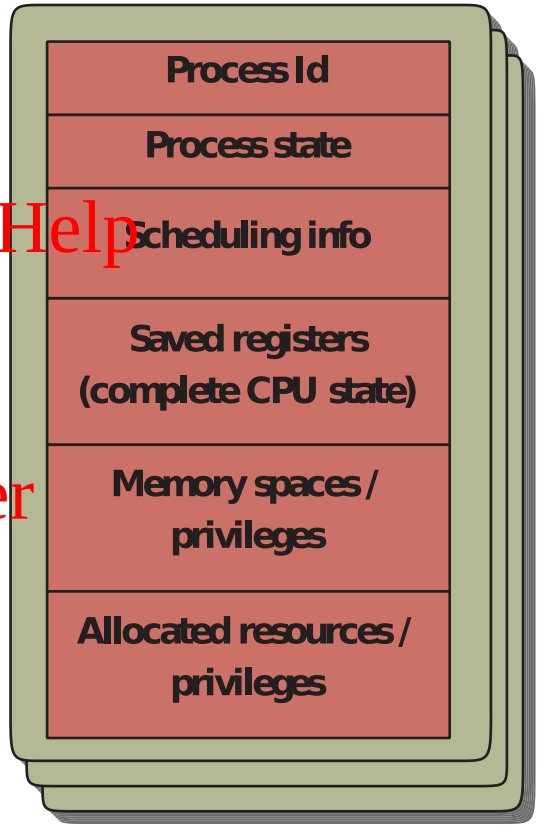
- Process Id
- Process state:
  {created, ready, executing, blocked, suspended, …}
- Scheduling attributes:
  Priorities, deadlines, consumed CPU-time, …
- CPU state:
  Saved/restored information on context switches (including program counter, stack pointer, …)



Process Control Blocks

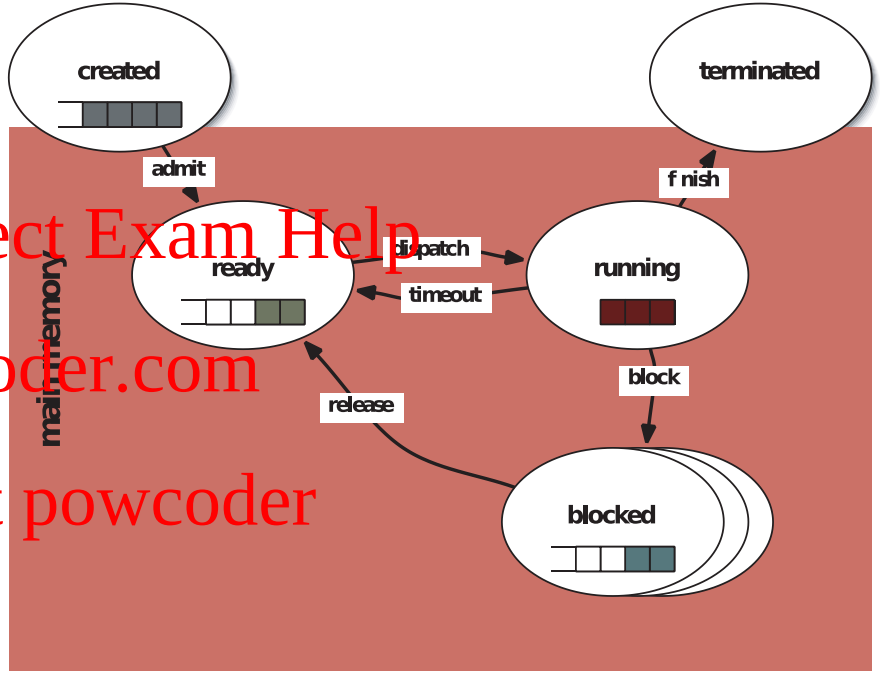
- …
- Memory attributes / privileges:
  memory base, limits, shared areas, …
- Allocated resources / privileges:
  open and requested devices and files, …

PCBs are commonly enqueued at a certain state or condition (awaiting access or change in state)



Process States

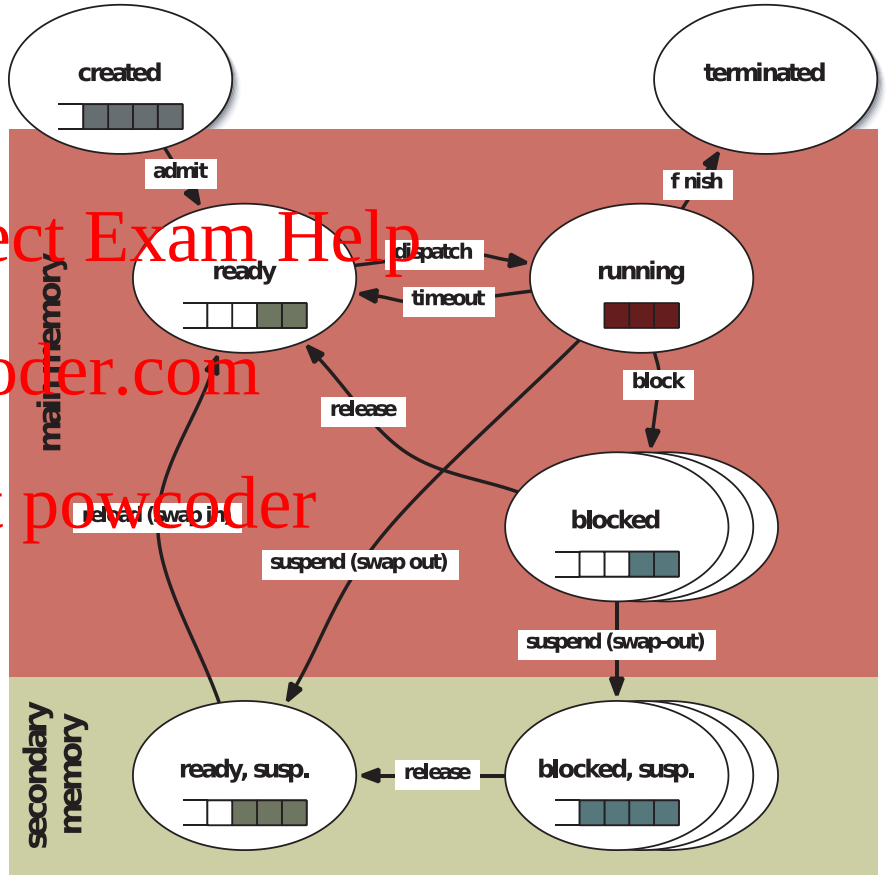
- **created**:
  task is ready to run, but not yet considered by any dispatcher (waiting for admission)
- **ready**:
  ready to run (waiting for a CPU)
- **running**:
  holding a CPU and executing
- **blocked**: not ready to run (waiting for a resource)



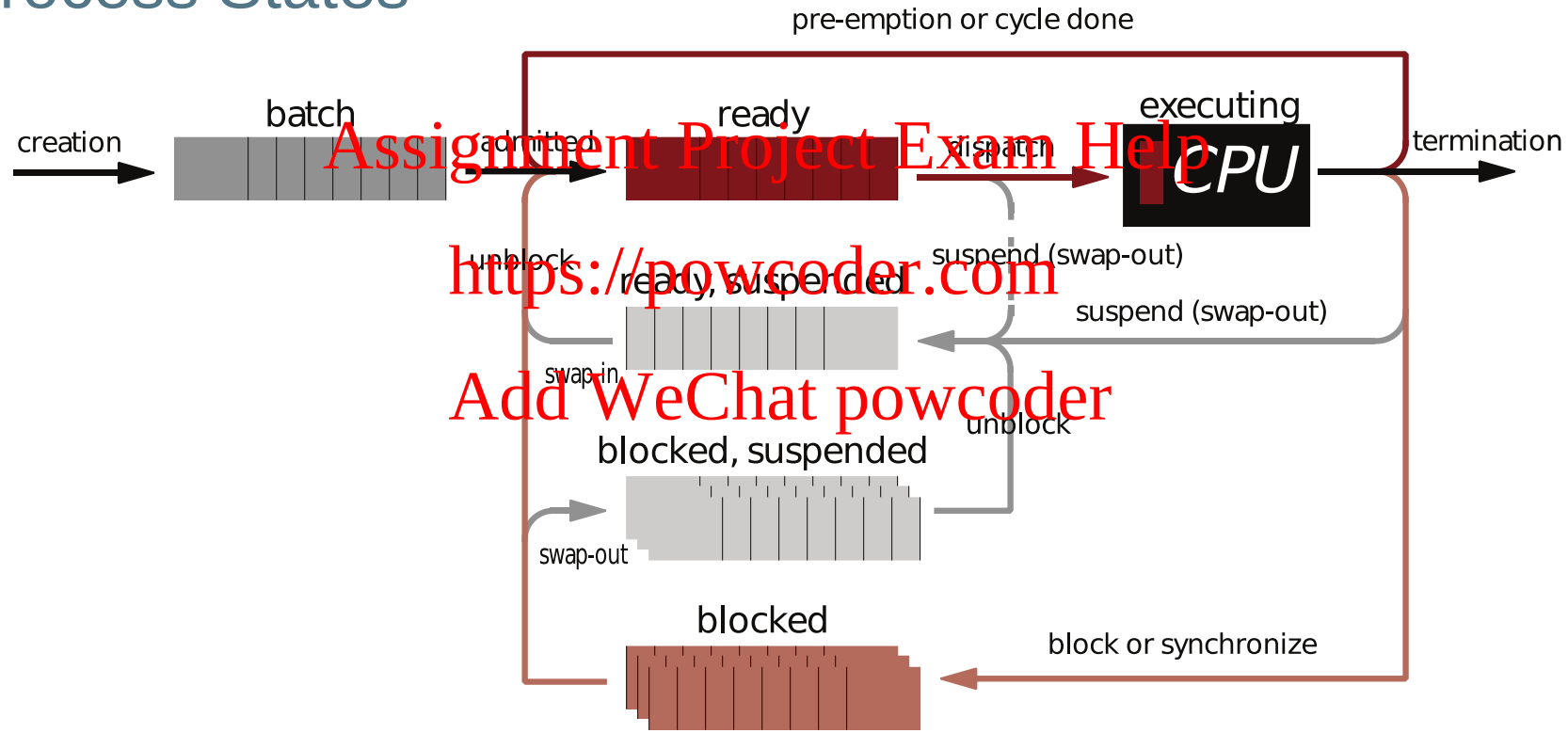
Process States

- **suspended** states: swapped out of main memory (non-time-critical processes) waiting for main memory space (or other resources)

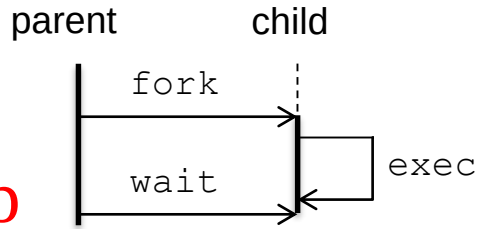
- dispatching and suspending can now be independent modules



Process States



Unix Processes



- In UNIX systems tasks are created by 'cloning'

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

  - duplicates the current process
  - returns 0 to the newly created process (the 'child' process)
  - returns the process id of child process to creating process (the 'parent' process)
  - or returns -1 as C-style indication of failure

```
if (fork() == 0) {
  // child process
  exec("path_to_exe", args);
  exit(0); // terminate
} else {
  // parent process
  …
  pid = wait();
  /* wait for termination of
one child process */
}
```

Communication Between Unix Processes

```
int pipe_fd[2], c, rc;
if (pipe(pipe_fd) == -1) {
  perror("no pipe"); exit(1);
}
if (fork() == 0) { // child/receiver
  close(pipe_fd[1]);
  while((rc = read(pipe_fd[0], &c,
1)) > 0) putchar(c);
  if (rc == -1) {
    perror("pipe broken");
    close(pipe_fd[0]); exit(1);
  }
  close(pipe_fd[0]);
}
```

```
else { // parent/sender
  close(pipe_fd[0]);
  while ((c = getchar()) > 0) {
    if (write(pipe_fd[1], &c, 1)== -1)
    {
      perror("pipe broken");
      close(pipe_fd[1]); exit(1);
    }
  }
  close(pipe_fd[1]);
  wait(NULL);
}
```