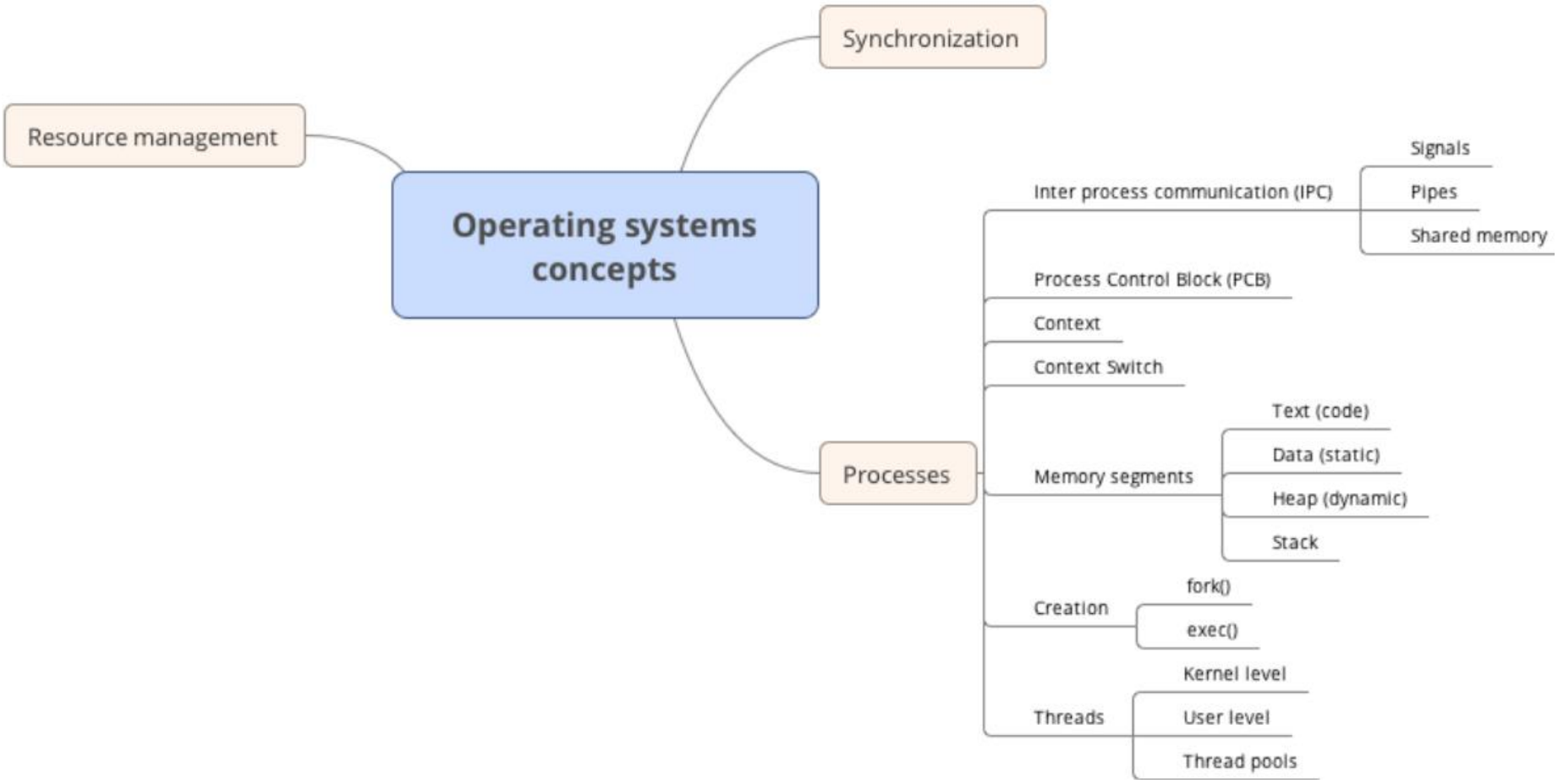
# CS341: Operating System

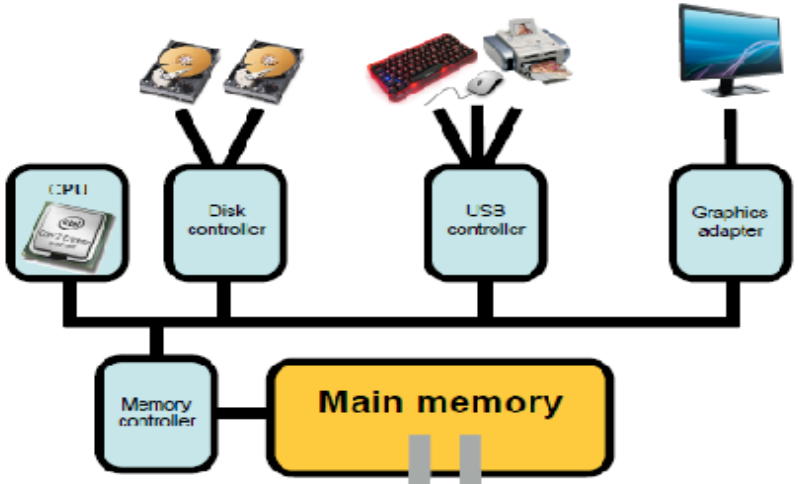
Course Files:

https://u.pcloud.link/publink/show?code=kZITAQXZCxtf3rGMIhpo90X7HMyGRXe3cBOk

Credit Some Slide: UCB CS162

# Process Management

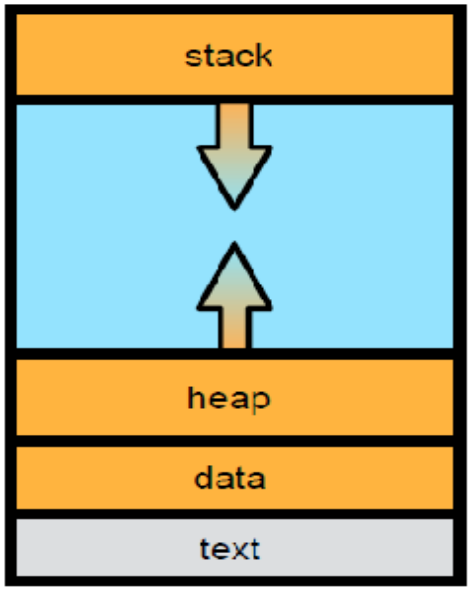

Operating systems concepts
Resource management
Synchronization
Processes
Inter process communication (IPC)
Signals
Pipes
Shared memory
Process Control Block (PCB)
Context
Context Switch
Memory segments
Text (code)
Data (static)
Heap (dynamic)
Stack
Creation
fork()
exec()
Threads
Kernel level
User level
Thread pools




## Process control block

- Process id (PID)
- Process state (new, ready, running, waiting or terminated)
- CPU Context
- I/O status information
- CPU scheduling information
- Memory management information

## Process memory image



# **From the Architecture to the OS to the User**: Architectural, resources, OS management, and User Abstractions.

| Hardware abstraction | Example OS Services | User abstraction |
|---|---|---|
| Processor | Process management, Scheduling, Traps, protection, accounting, synchronization | Process |
| Memory | Management, Protection, virtual memory | Address spaces |
| I/O devices | Concurrency with CPU, Interrupt handling | Terminal, mouse, printer, system calls |
| File System | File management, Persistence | Files |
| Distributed systems | Networking, security, distributed file system | Remote procedure calls, network file system |

# CPU starts…

CPU starts and loads instructions starting at 0xffffffff0

- Instruction jumps to BIOS code
- BIOS (Basic Input/Output System) is started

– Performs basic tests (memory, keyboard, etc) – POST (power on self test)

– Determines the “boot device” (Hard disk, Floppy, CD-ROM)

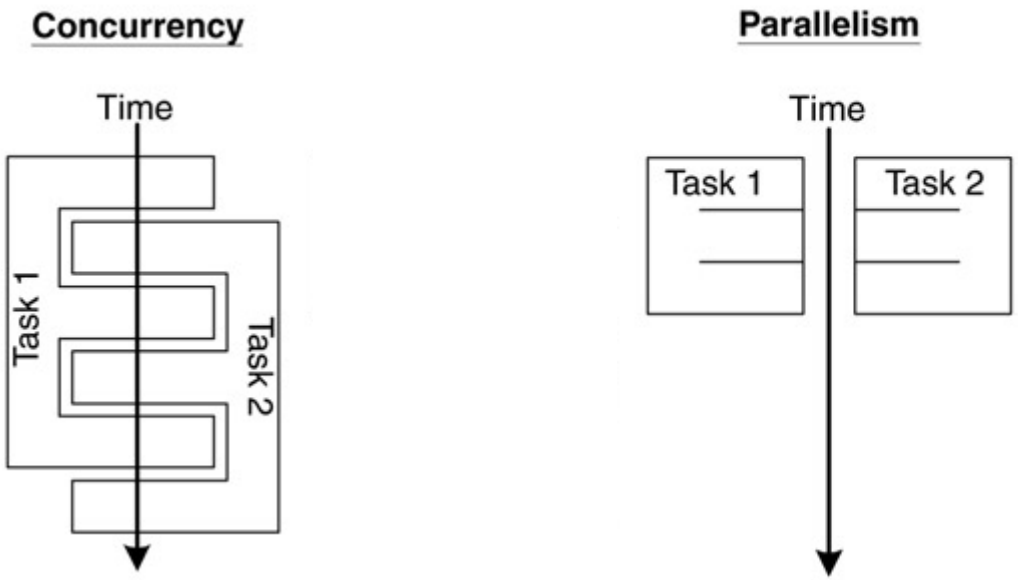
– Loads the contents of the first physical sector (the Master Boot Record -
MBR - Cyl 0, Head 0, Sect 1) in memory 0x7C00 - 0x7DFF

– Jumps to 0x7C00

- MBR code finds an “active” file system, loads the corresponding boot sector in memory, and jumps to it
- The boot sector code loads the *operating system*

# Concurrency and Parallelism

- Concurrent multithreading systems **give the appearance of several tasks executing at once**, but these tasks are actually split up into chunks that share the processor with chunks from other tasks.
- In parallel systems, two tasks are actually performed simultaneously. **Parallelism requires a multi-CPU system.**

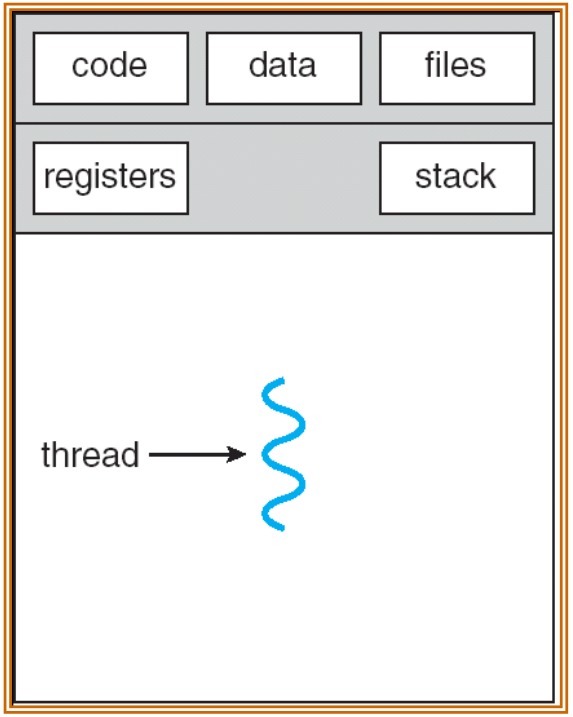


# Process Management

- A process as the unit of execution.
- How are processes represented in the OS?
- What are possible execution states and how does the system move from one state to another?
- How are processes created in the system?
- How do processes communicate? Is this efficient?

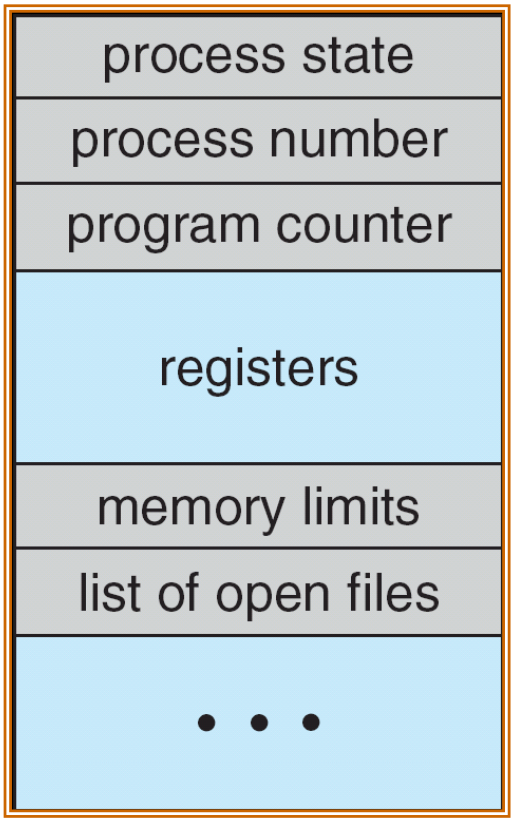
# Starting Point: Single Threaded Process

- Process: OS abstraction of what is needed to run a single program
  1. Sequential program execution stream
     - Sequential stream of execution (thread)
     - State of CPU registers
  2. Protected resources
     - Contents of Address Space
     - I/O state



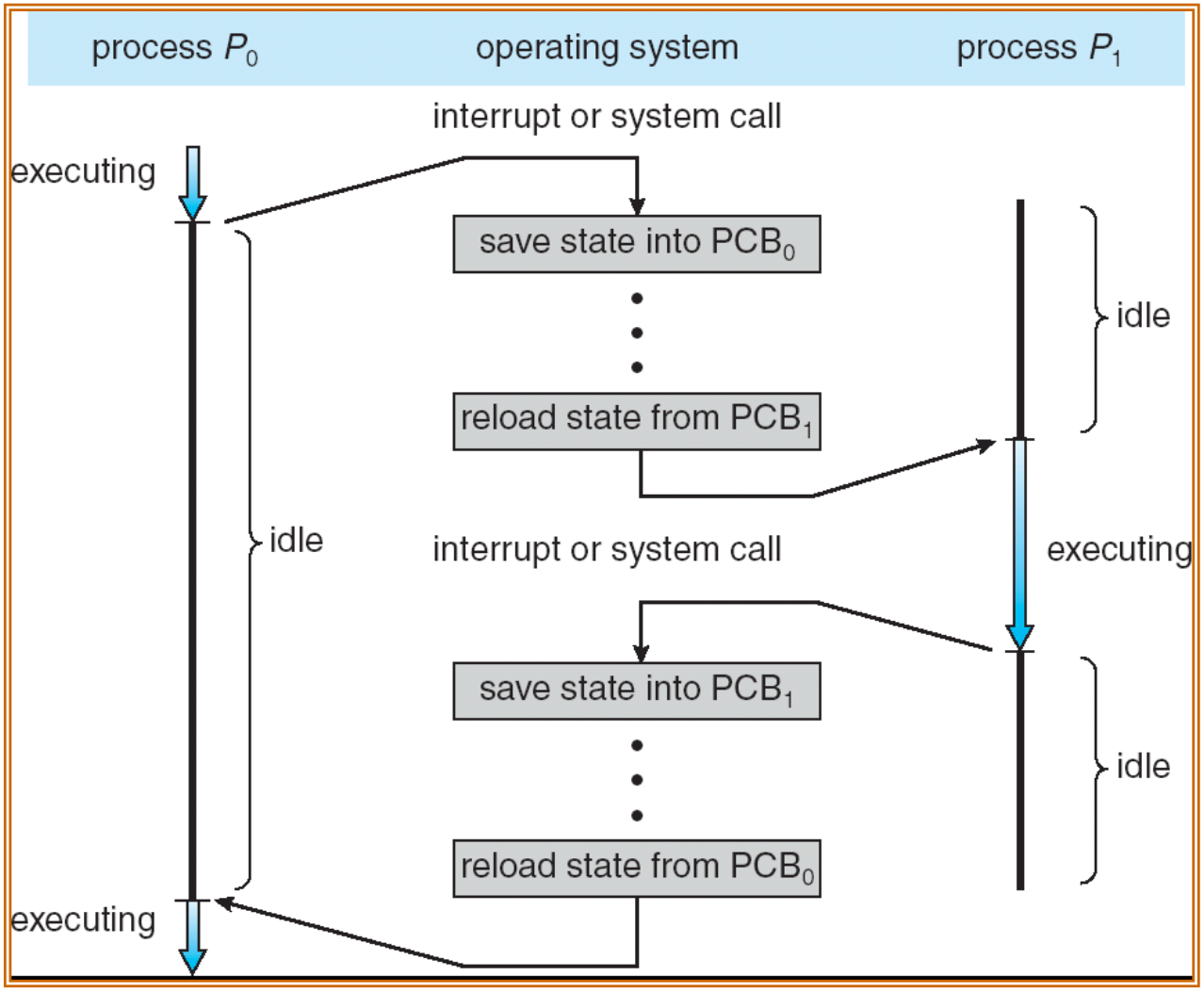
# Multiplexing Processes

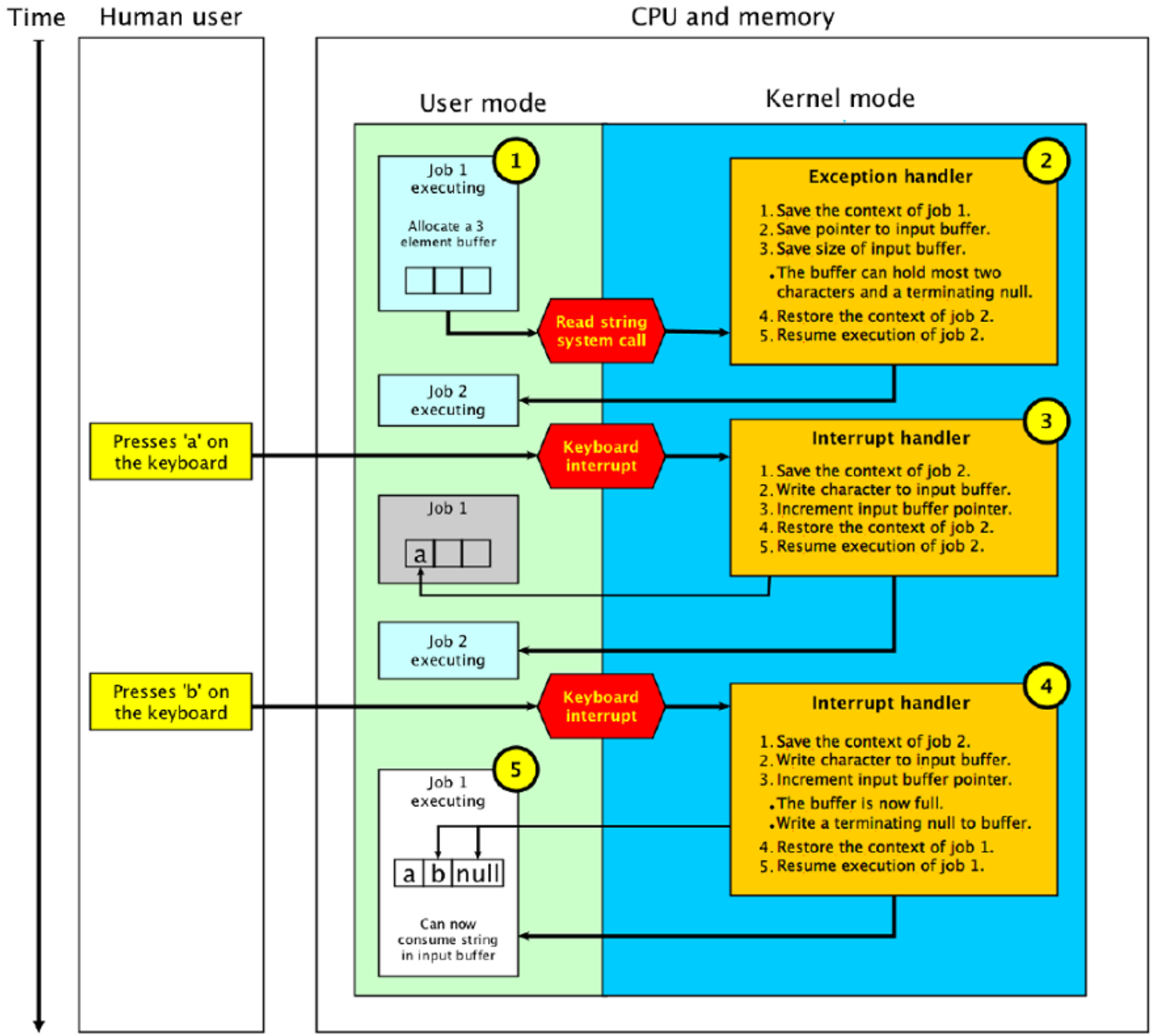
- Snapshot of each process in its PCB
  - Only one active at a time (per core…)
- Give out CPU to different processes
  - **Scheduling**
  - **Policy Decision**
- Give out non-CPU resources
  - Memory/IO
  - Another **policy decision**




Process Control Block

# Context Switch




Time
Human user
CPU and memory
User mode
Kernel mode
Job 1 executing
Allocate a 3 element buffer
1
Read string system call
Exception handler
1. Save the context of job 1.
2. Save pointer to input buffer.
3. Save size of input buffer.
• The buffer can hold most two characters and a terminating null.
4. Restore the context of job 2.
5. Resume execution of job 2.
2
Job 2 executing
Presses 'a' on the keyboard
Keyboard interrupt
Interrupt handler
1. Save the context of job 2.
2. Write character to input buffer.
3. Increment input buffer pointer.
4. Restore the context of job 2.
5. Resume execution of job 2.
3
Job 1
a
Job 2 executing
Presses 'b' on the keyboard
Keyboard interrupt
Interrupt handler
1. Save the context of job 2.
2. Write character to input buffer.
3. Increment input buffer pointer.
• The buffer is now full.
• Write a terminating null to buffer.
4. Restore the context of job 1.
5. Resume execution of job 1.
4
Job 1 executing
a b null
Can now consume string in input buffer
5

# Context Switch

Starting and stopping processes is called a **context switch**, and is a relatively expensive operation.

• The OS starts executing a ready process by loading hardware registers (PC, SP, etc) from its PCB

• While a process is running, the CPU modifies the Program Counter (PC), Stack Pointer (SP), registers, etc.

• When the OS stops a process, it saves the current values of the registers, (PC, SP, etc.) into its PCB

• This process of switching the CPU from one process to another (stopping one and starting the next) is the context switch.

  – Time sharing systems may do 100 to 1000 context switches a second.
  – The cost of a context switch and the time between switches are closely related

# Creating a Process

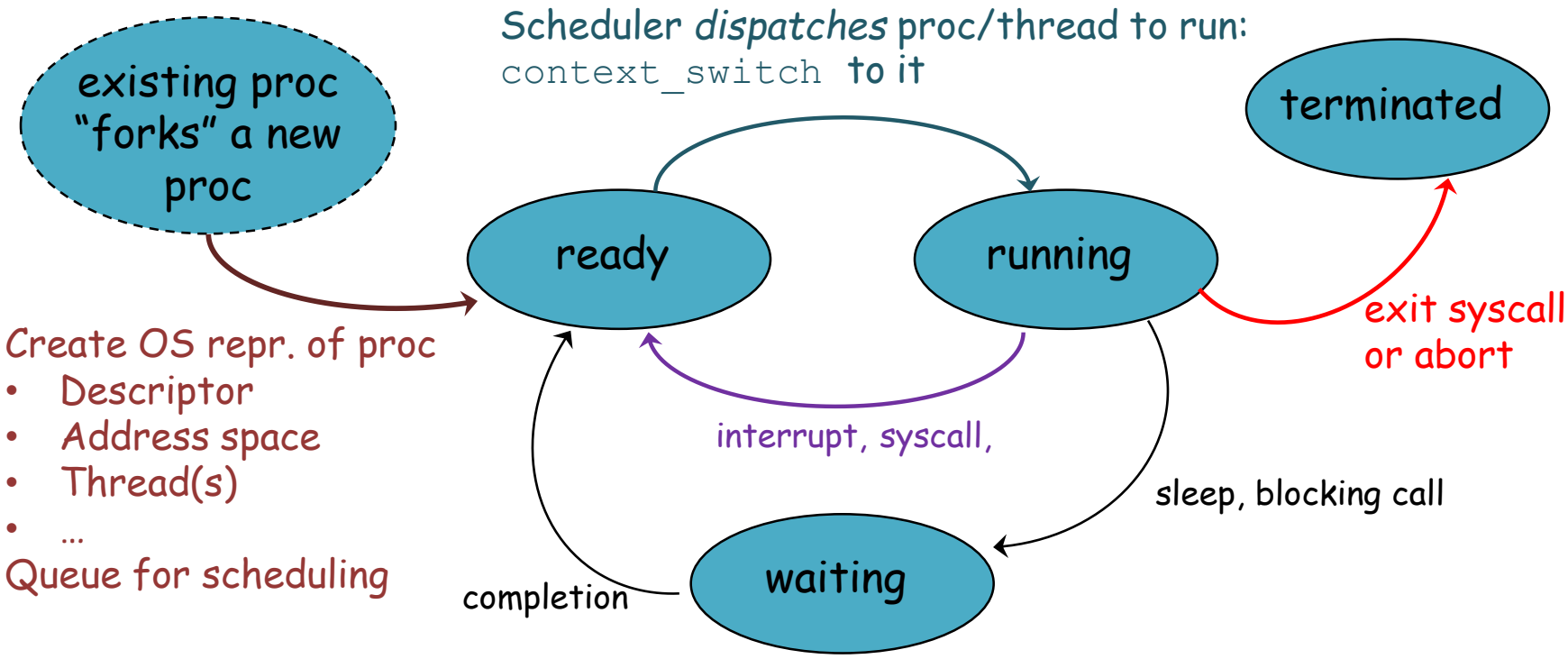
**One process can create other processes to do work.**

- The creator is called the *parent* and the new process is the *child*
- The parent defines (or donates) resources and privileges to its children
- A parent can either wait for the child to complete, or continue in parallel
- In Unix, the *fork* system call called is used to create child processes
- Fork copies variables and registers from the parent to the child
- The *only difference* between the child and the parent is the value returned by fork

* In the parent process, fork returns the process id of the child
* In the child process, the return value is 0

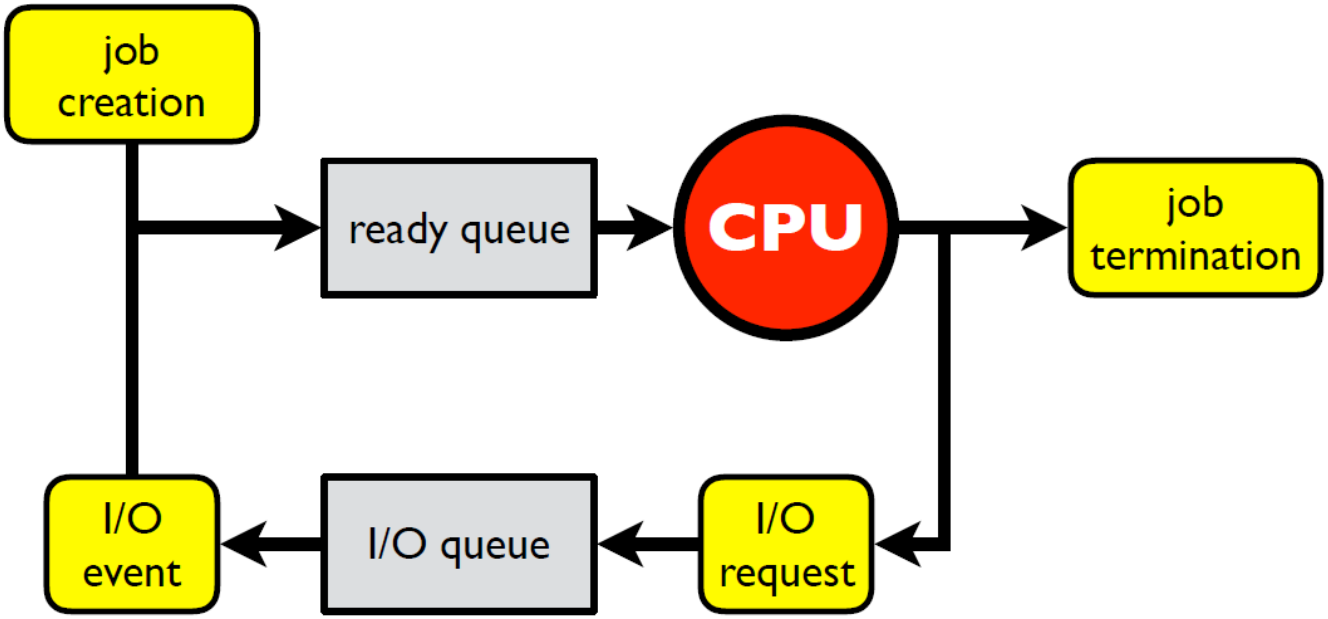
- The parent can wait for the child to terminate by executing the *wait* system call or continue execution
- The child often starts a new and different program within itself, via a call to *exec* system call.

# Lifecycle of a process




- OS juggles many process/threads using kernel data structures
- Proc's may create other process (fork/exec)
  - All starts with init process at boot

Pintos: process.c


job creation
ready queue
CPU
job termination
I/O event
I/O queue
I/O request

# Process Execution State

Execution state of a process indicates what it is doing

*new:* the OS is setting up the process state

*running:* executing instructions on the CPU

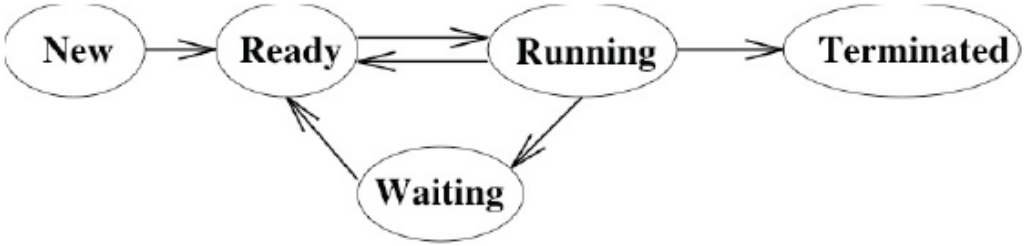
*ready:* ready to run, but waiting for the CPU

*waiting:* waiting for an event to complete

*terminated:* the OS is destroying this process

- As the program executes, it moves from state to state, as a result of the program actions (e.g., system calls), OS actions (scheduling), and external actions (interrupts).

# Process Execution State




- **Example:**

```
void main() {
  printf('Hello World');
}
```

| **state sequence** |
| --- |
| new |
| ready |
| running |
| waiting for I/O |
| ready |
| running |
| terminated |

- The OS manages multiple active process using *state queues* (More on this in a minute…)

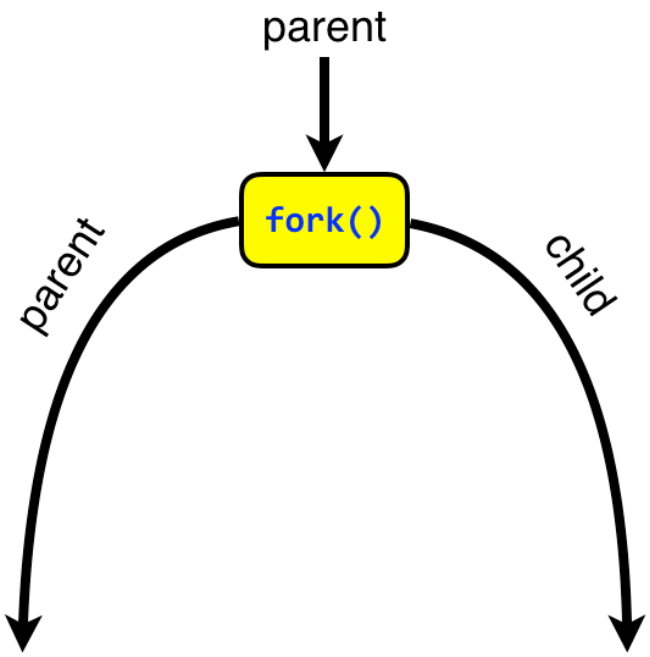
# Creating a Process: Example

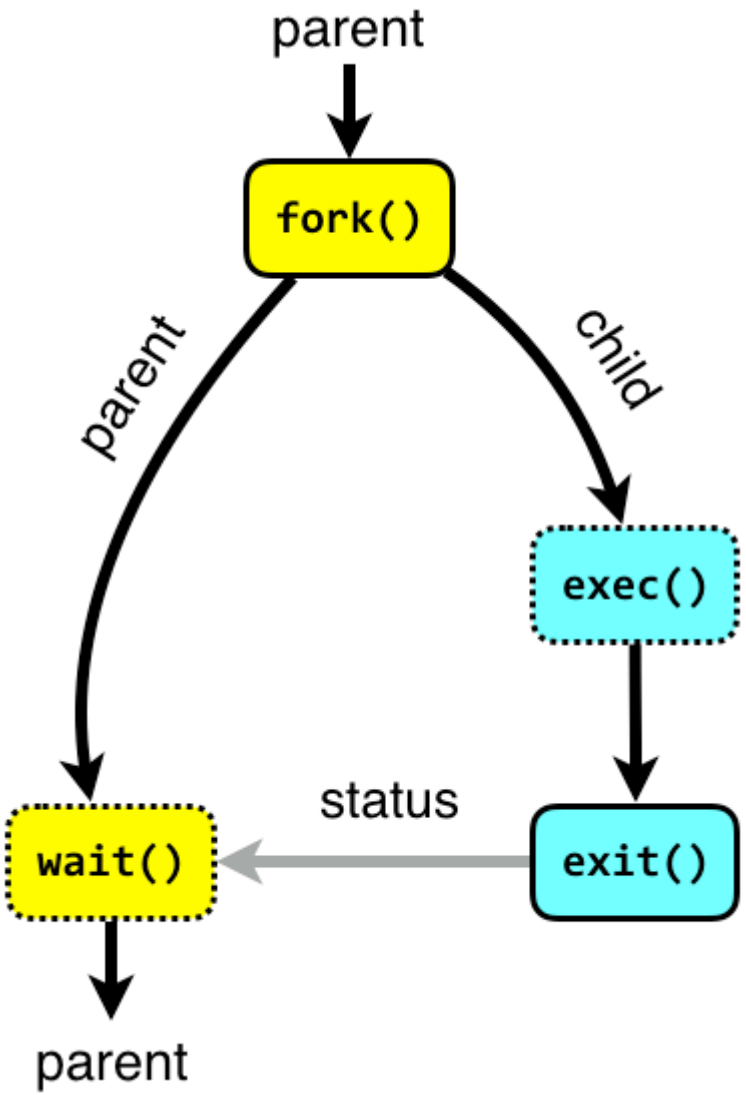
When you log in to a machine running Unix, you create a shell process.

- Every command you type into the shell is a child of your shell process and is an implicit *fork* and *exec* pair.

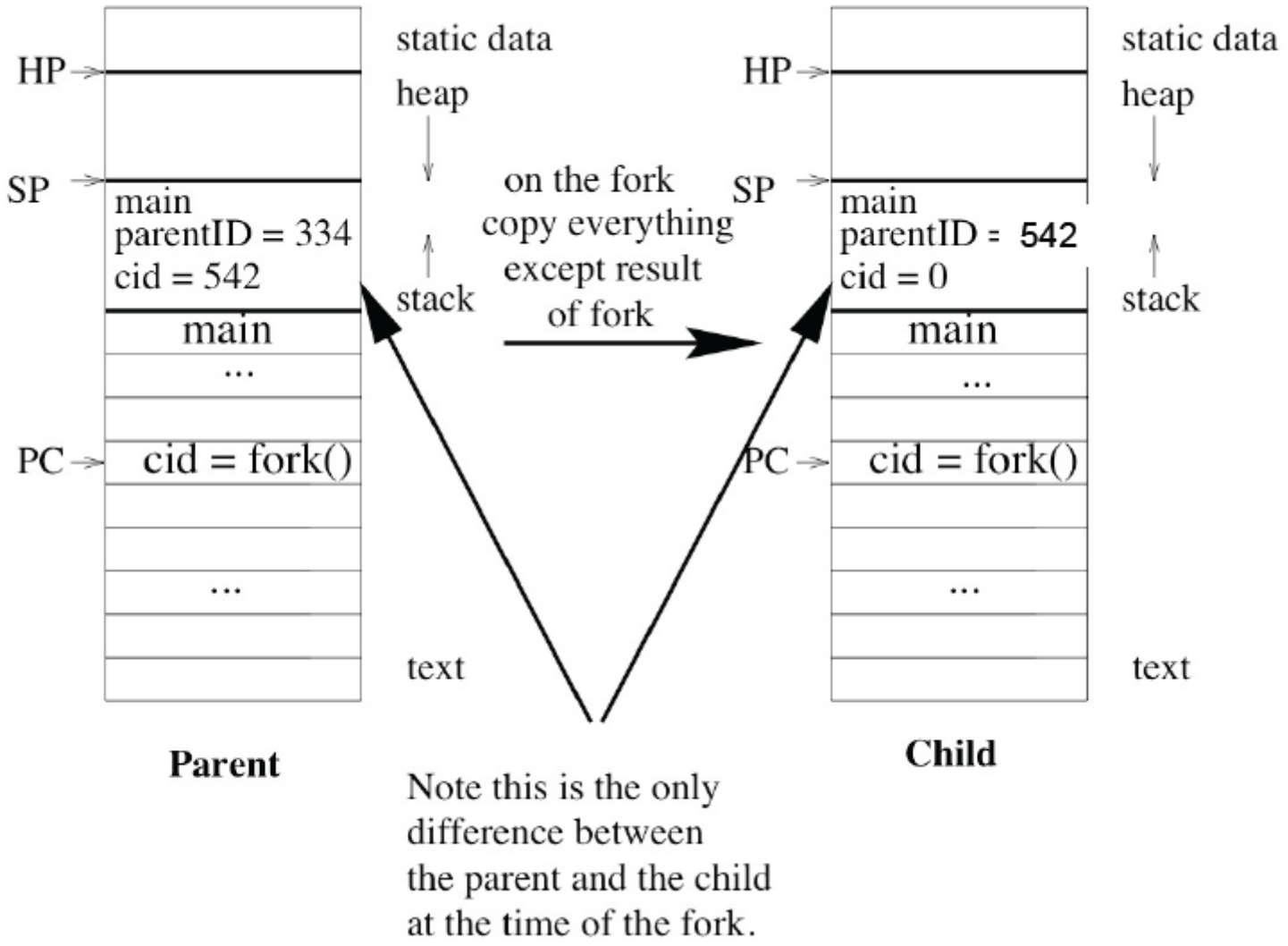
- For example, you type emacs, the OS "*forks*" a new process and then "*exec*" (executes) emacs.

- If you type an & after the command, Unix will run the process in parallel with your shell, otherwise, your next shell command must wait until the first one completes.


parent
fork()
parent
child


parent
fork()
parent
child
exec()
status
wait()
exit()
parent

# What is happening on the Fork



# Example Unix Program: Explanation

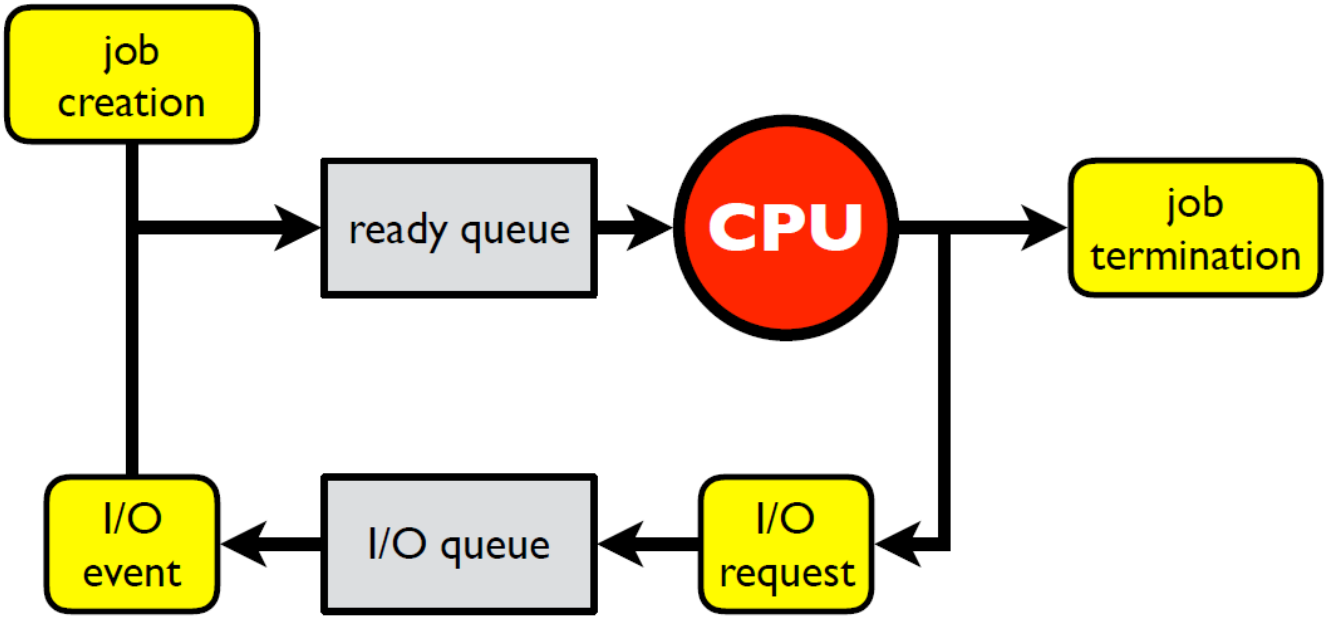
**fork()** forks a new child process that is a copy of the parent.

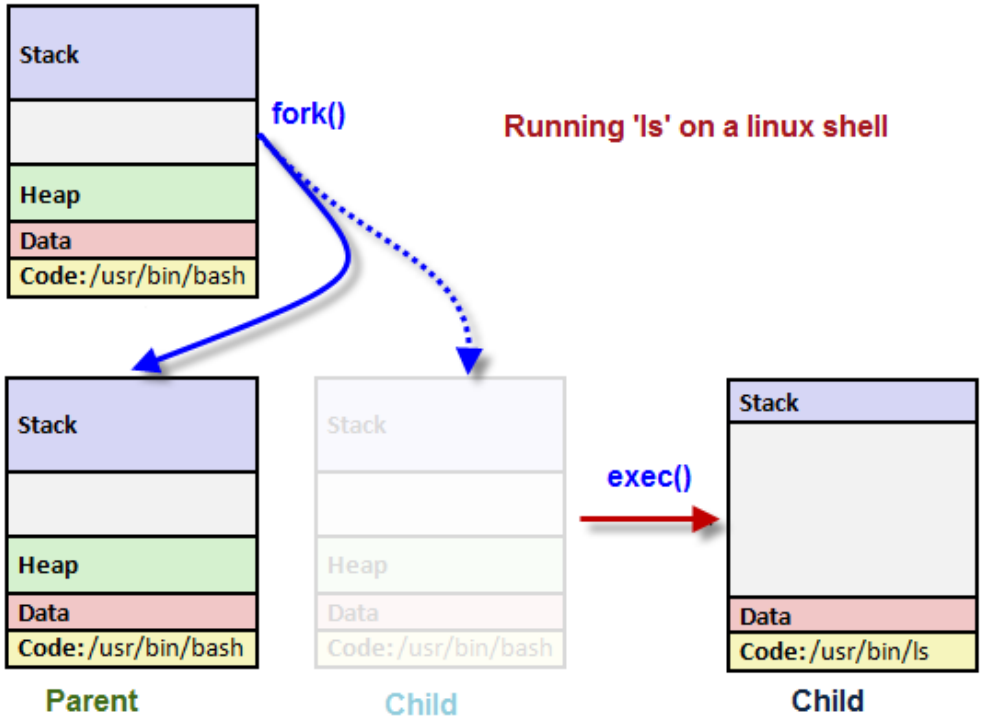
**execlp()** replaces the program of the current process with the named program.

**sleep()** suspends execution for at least the specified time.

**waitpid()** waits for the named process to finish execution.

**gets()** reads a line from a file.


job creation
ready queue
CPU
job termination
I/O event
I/O queue
I/O request


Stack
Heap
Data
Code: /usr/bin/bash
fork()
Running 'ls' on a linux shell
Stack
Heap
Data
Code: /usr/bin/bash
Parent
Stack
Heap
Data
Code: /usr/bin/bash
Child
exec()
Stack
Data
Code: /usr/bin/ls
Child

# Example Program: Fork

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char ** argv)
{
   int pid;

   printf("Before forking in the parent\n");

   pid = fork();
   if (pid == 0)
      printf("I am the child.\n");
   else
      printf("I am the proud parent of pid %d\n", pid);

   return 0;
}
```

# Example Program: Fork

```
#!/usr/bin/env python2

import os

print 'Before forking in the parent'

pid = os.fork()
if pid == 0:
   print 'I am the child'
else:
   print 'I am the proud parent of pid', pid
```

# Example Program: Fork

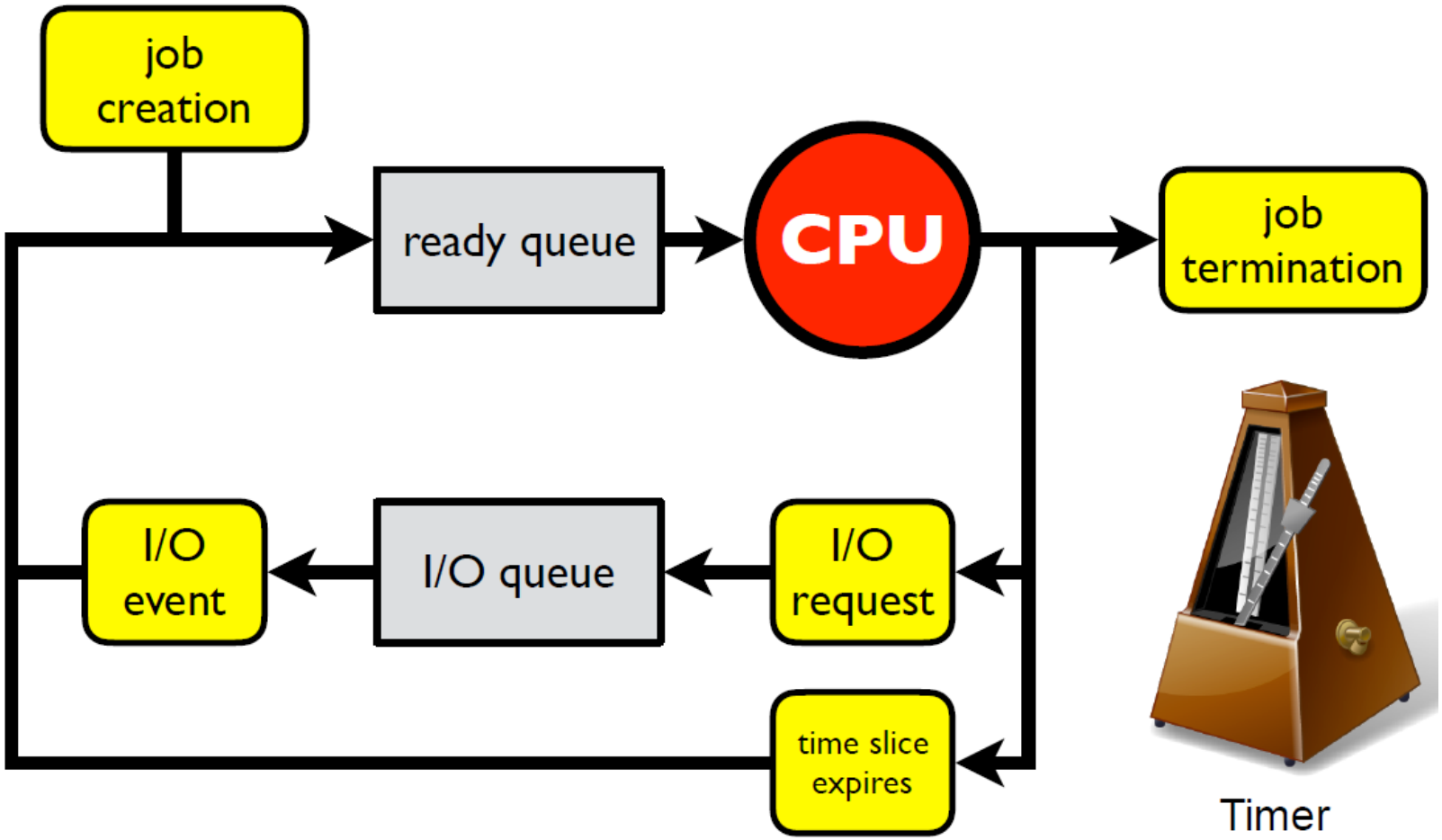
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char ** argv)
{
   int pid;

   pid = fork();
   if (pid == 0)
      execl("/bin/echo", "/bin/echo", "hello", (char *) NULL);

   printf("Only the parent gets here\n");

   return 0;
}
```

# Example Unix Program: Fork

```
#include <sys/wait.h>
#include <stdio.h>
main() {
int parentID = getpid(); /* ID of this process */
char prgname[1024];
gets(prgname); /* read the name of program we want to start */
int cid = fork();
if(cid == 0) { /* I'm the child process */
execlp( prgname, prgname, 0); /* Load the program */
/* If the program named prgname can be started, we never get
to this line, because the child program is replaced by prgname */
printf("I didn't find program %s\n", prgname);
} else { /* I'm the parent process */
sleep (1); /* Give my child time to start. */
waitpid(cid, 0, 0); /* Wait for my child to terminate. */
printf("Program %s finished\n", prgname);
} }
```

| **Function** | **Specification of program file** *(–, p)* | **Specification of arguments** *(v, l)* | **Source of environment** *(e, –)* |
|---|---|---|---|
| *execve()* | pathname | array | *envp* argument |
| *execle()* | pathname | list | *envp* argument |
| *execlp()* | filename + PATH | list | caller's *environ* |
| *execvp()* | filename + PATH | array | caller's *environ* |
| *execv()* | pathname | array | caller's *environ* |
| *execl()* | pathname | list | caller's *environ* |

# A schematic view of multitasking



# A schematic view of multiprogramming