Processes

Abhijit A M
abhijit.comp@coep.ac.in

Process related data structures in kernel code

- Kernel needs to maintain following types of data structures for managing processes
  - List of all processes
  - Memory management details for each, files opened by each etc.
  - Scheduling information about the process
  - Status of the process
  - List of processes “waiting” for different events to occur,
  - Etc.

Process Control Block

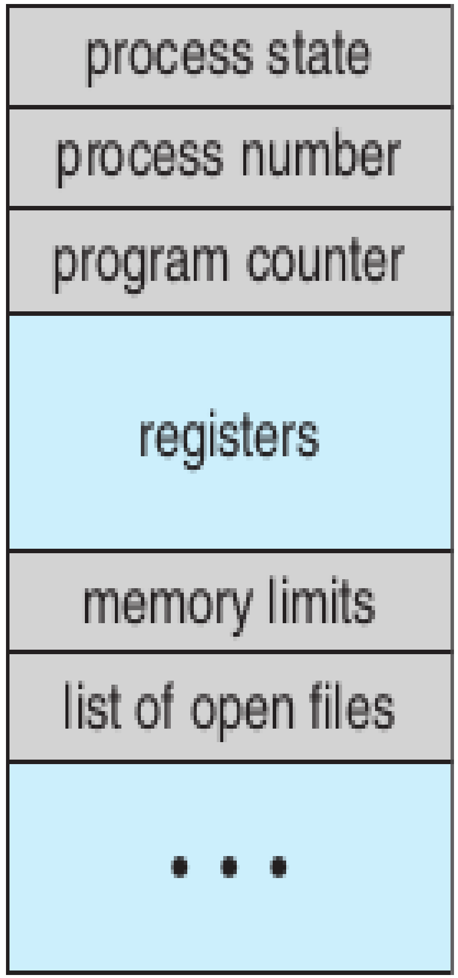


Figure 3.3 Process control block (PCB).

– A record representing a process in operating system’s data structures

– OS maintains a “list”of PCBs, one for each process

– Called “`struct task_struct`” in Linux kernel code and “`struct proc`” in xv6 code

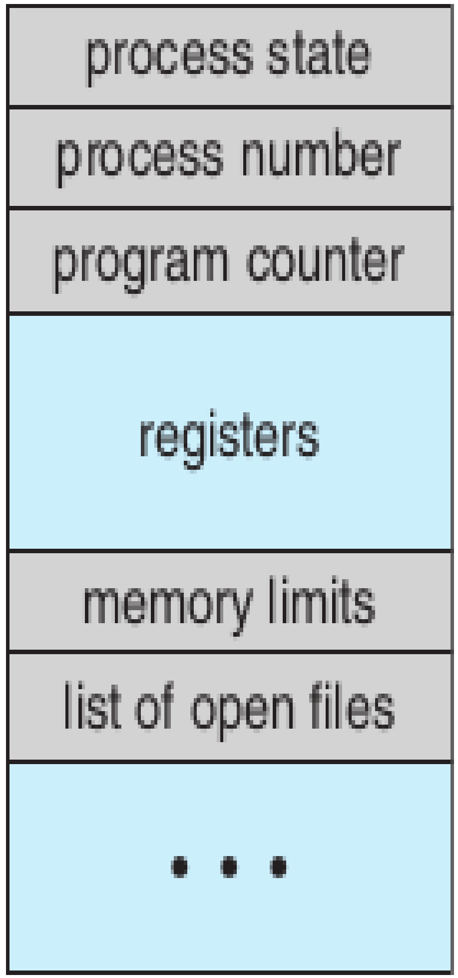


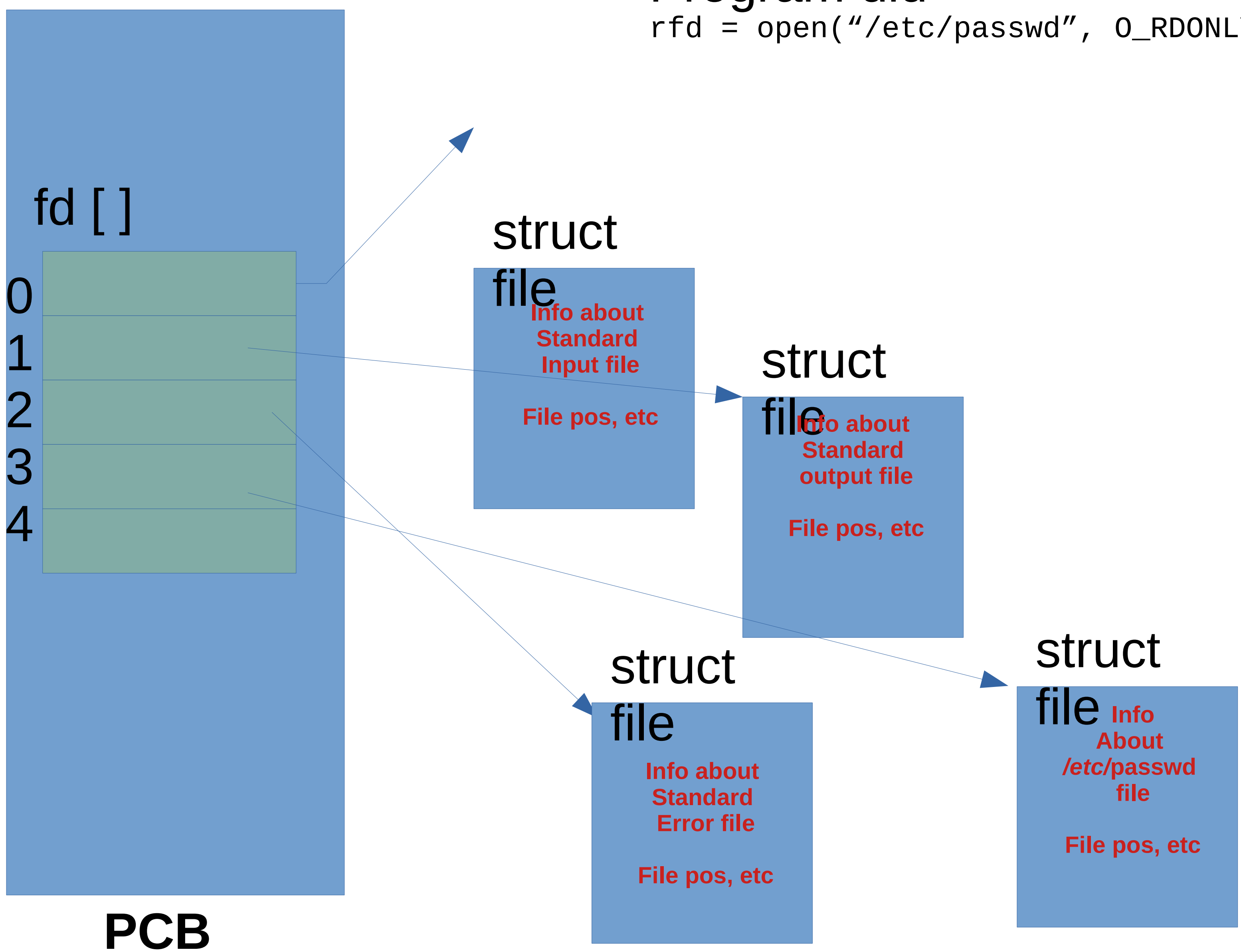
Figure 3.3 Process control block (PCB).

Fields in PCB

- Process ID (PID)
- Process State
- Program counter
- Registers
- Memory limits of the process
- Accounting information
- I/O status
- Scheduling information
- array of file descriptors (list of open files)
- ...etc

List of open files

Program did
`rfd = open(“/etc/passwd”, O_RDONLY)`



List of open files

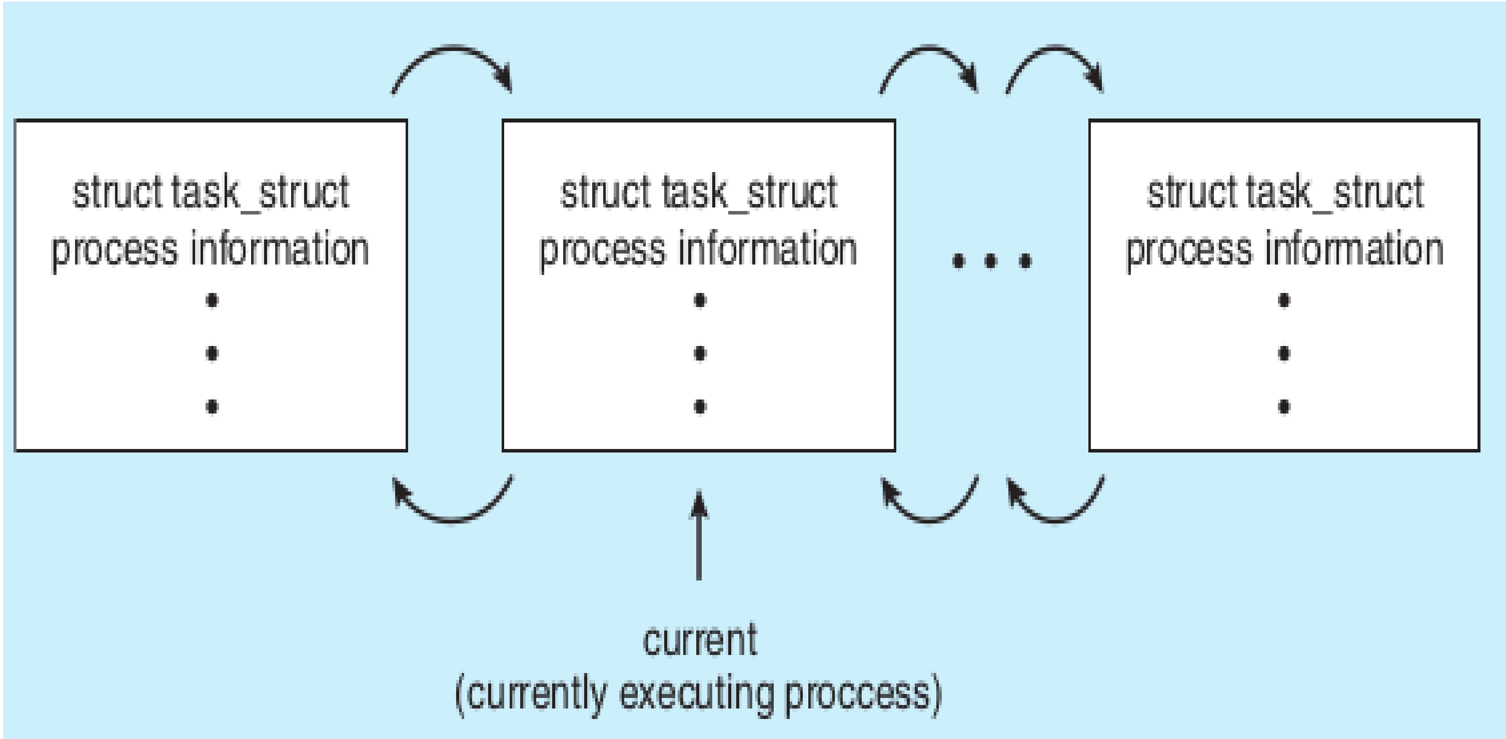
- The PCB contains an array of pointers, called file descriptor array (fd[ ]), pointers to structures representing files
- When open() system call is made
  - A new file structure is created and relevant information is stored in it
  - Smallest available of fd [ ] pointers is made to point to this new struct file
  - The index of this fd [ ] pointer is returned by open
- When subsequent calls are made to read(fd, ...) or write(fd, ...) , etc.
  - The kernel gets the “fd” as an index in the fd[ ] array and is able to locate the file structure for that file

```
// XV6 Code : Per-process state
enum procstate { UNUSED, EMBRYO, SLEEPING,
RUNNABLE, RUNNING, ZOMBIE };

struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this
process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};

struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
```

```
struct file {
  enum { FD_NONE,
FD_PIPE, FD_INODE } type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe;
  struct inode *ip;
  uint off;
};
```

Process Queues/Lists inside OS

- Different types of queues/lists can be maintained by OS for the processes
  - A queue of processes which need to be scheduled
  - A queue of processes which have requested input/output to a device and hence need to be put on hold/wait
  - List of processes currently running on multiple CPUs
  - Etc.




```
// Linux data structure
struct task_struct {
    long state;/*state of the process */
    struct sched_entity se; /* scheduling information */
    struct task_struct *parent; /*this process's parent */
    struct list_head children; /*this process's children */
    struct files_struct *files; /* list of open files */
    struct mm_struct *mm;/*address space */
```

```
struct list_head {
    struct list_head
*next, *prev;
};
```

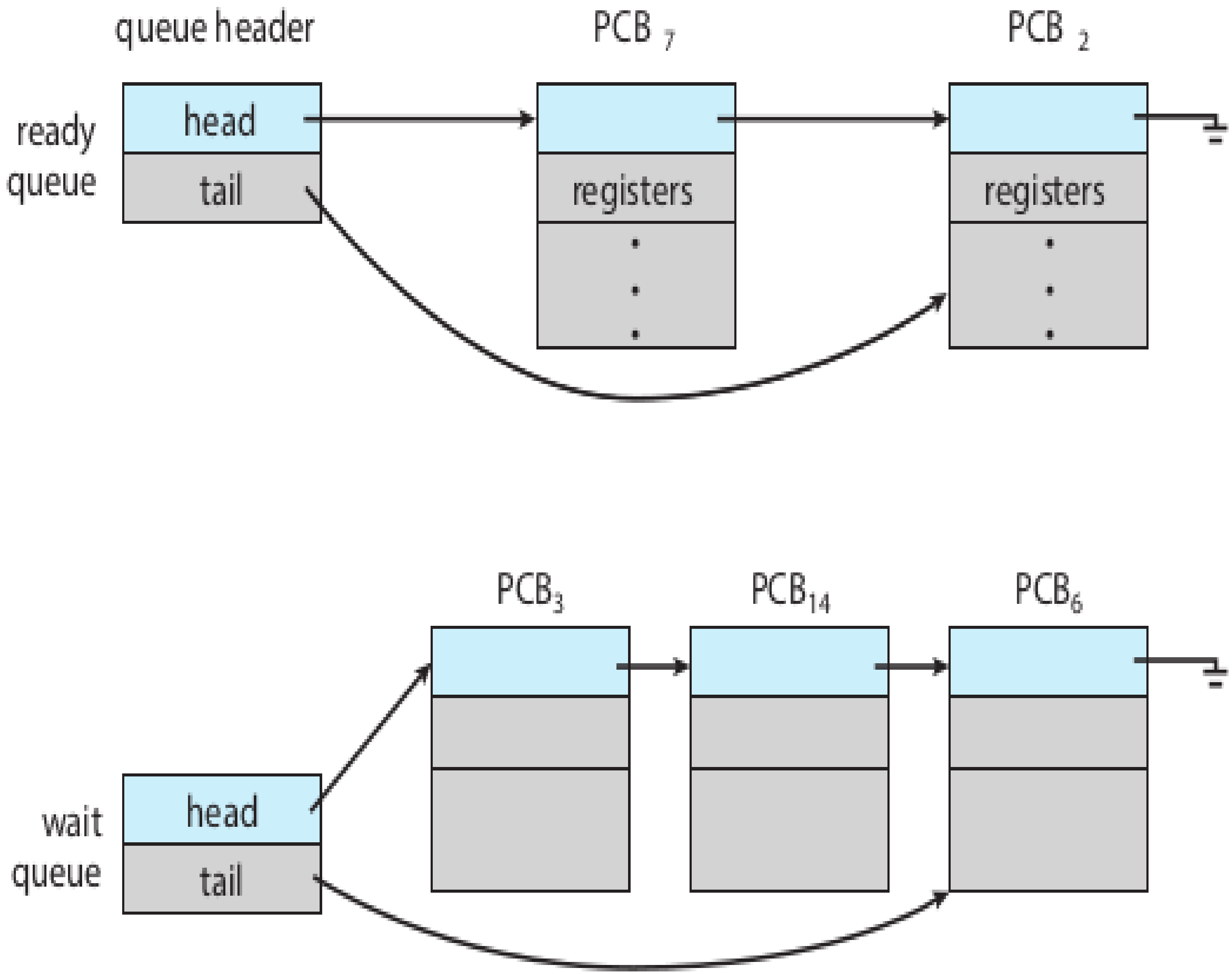


Figure 3.4 The ready queue and wait queues.

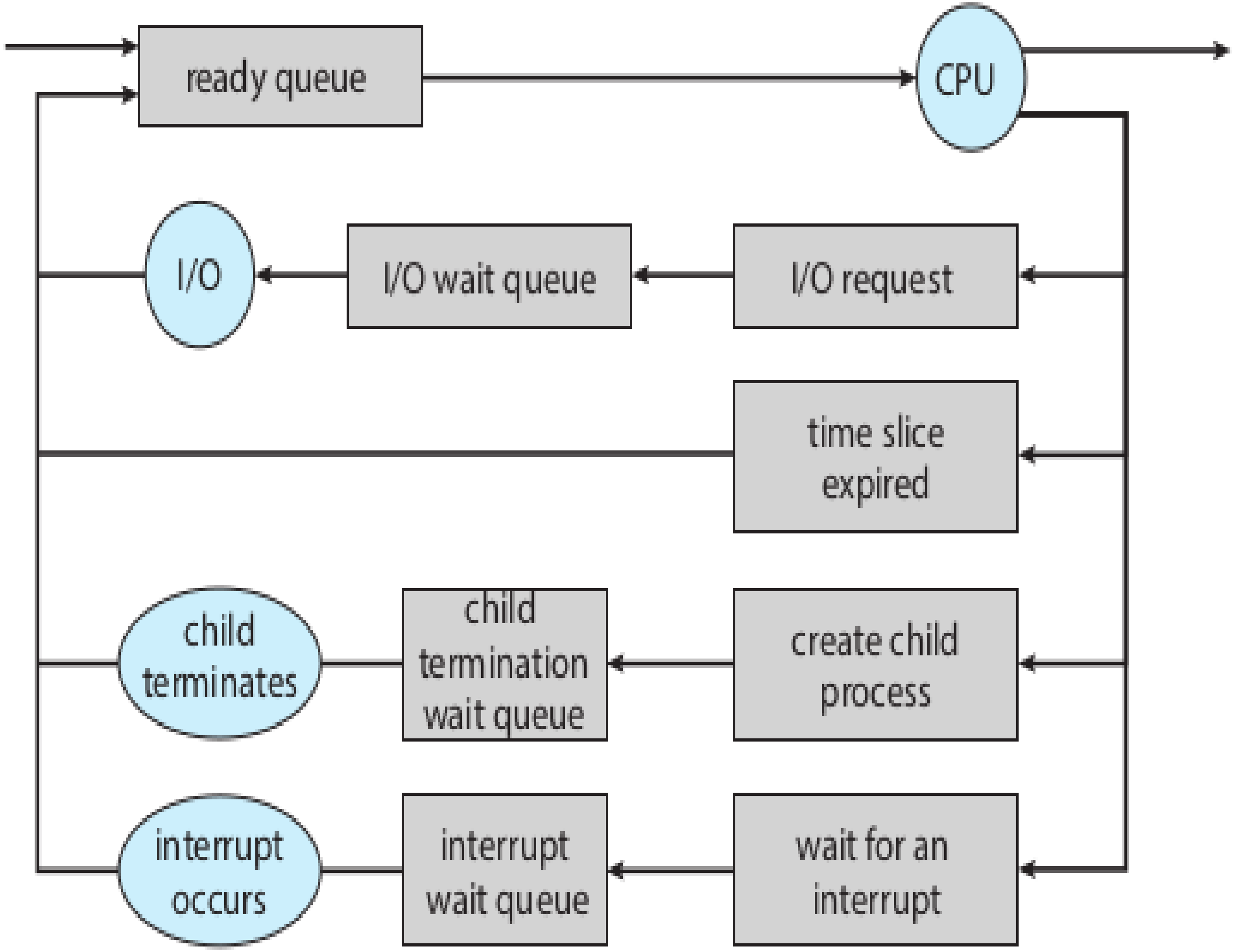


Figure 3.5 Queueing-diagram representation of process scheduling.

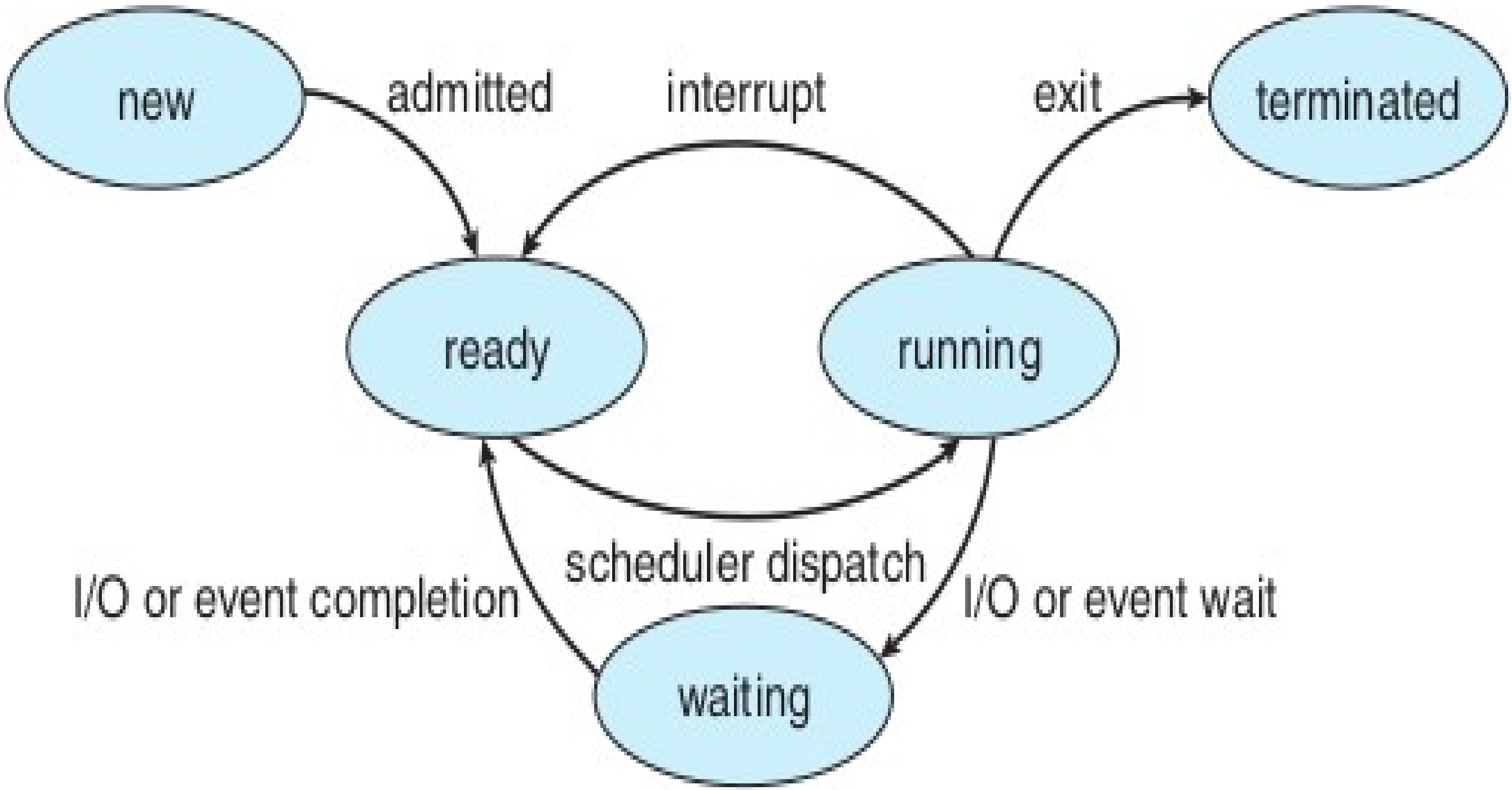


Figure 3.2 Diagram of process state.

Conceptual diagram

“Giving up” CPU by a process or blocking

```
int main() {
i = j + k;
scanf("%d", &k);
}
int scanf(char *x, ...) {
...
read(0, ..., ...);
}
int read(int fd, char *buf, int len) {
...
__asm__ { "int 0x80..."}
...
}
```

OS Syscall

```
sys_read(int fd, char *buf, int len) {
 file f  = current->fdarray[fd];
 int offset = f->position;
 ...
 disk_read(... , offset, ...);
 // Do what now?
 //asynchronous read
 //Interrupt will occur when the disk read is complete
 // Move the process from ready queue to a wait queue and call scheduler!
 // This is called "blocking"
 Return the data read ;
}
disk_read(...., offset, .... ) {
__asm__("outb PORT ..");
return;
}
```

“Giving up” CPU by a process or blocking

The relevant code in xv6 is in

Sleep()

The wakeup code is in wakeup() and wakeup1()

To be seen later

Context Switch

- Context
  - Execution context of a process
  - CPU registers, process state, memory management information, all configurations of the CPU that are specific to execution of a process/kernel
- Context Switch
  - Change the context from one process/OS to OS/another process
  - Need to save the old context and load new context
  - Where to save? --> PCB of the process

Context Switch

- Is an overhead
- No useful work happening while doing a context switch
- Time can vary from hardware to hardware
- Special instructions may be available to save a set of registers in one go

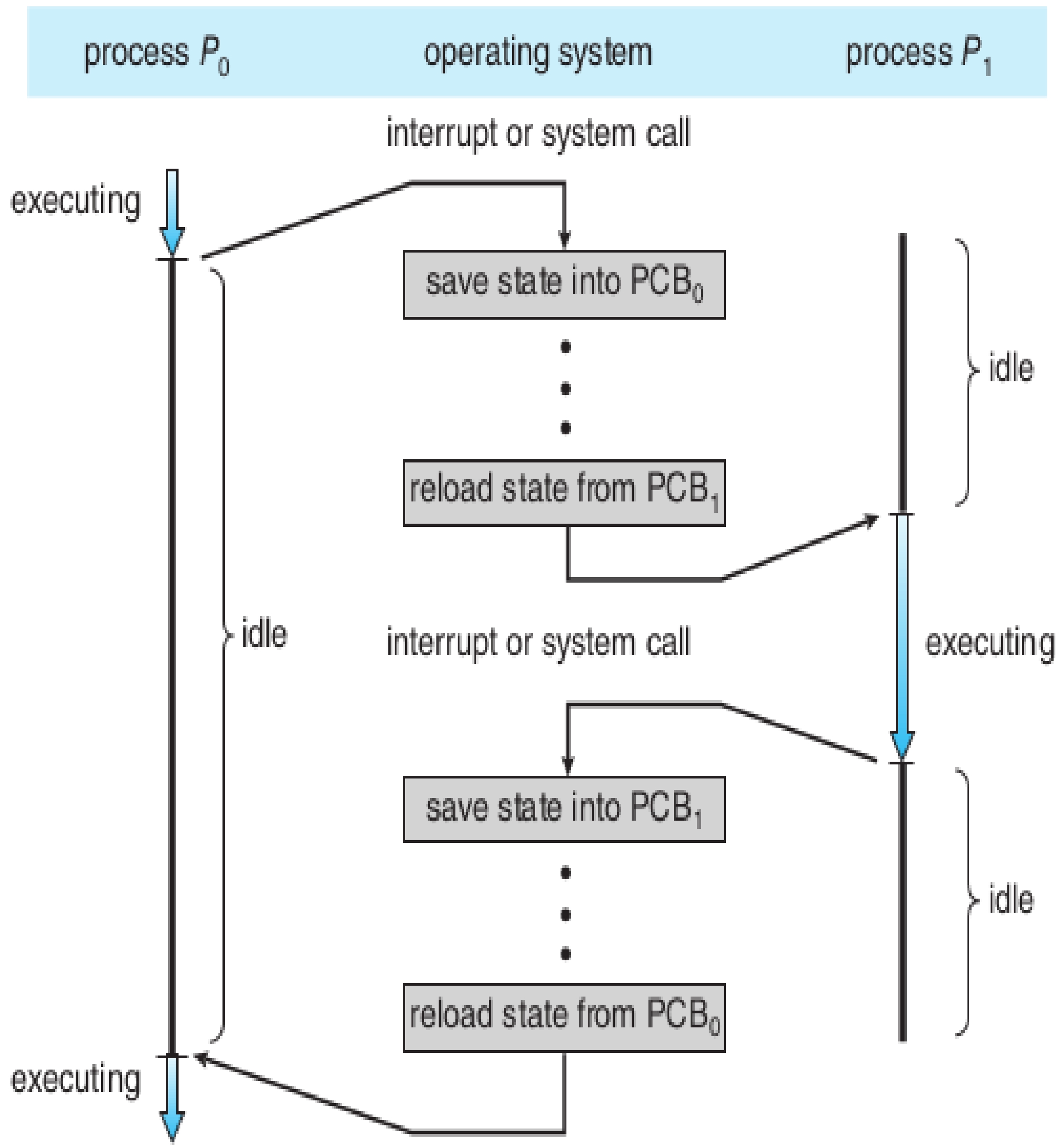


Figure 3.6 Diagram showing context switch from process to process.

Pecularity of context switch

- When a process is running, the function calls work in LIFO fashion
  - Made possible due to calling convention
- When an interrupt occurs
  - It can occur anytime
  - Context switch can happen in the middle of execution of any function
- After context switch
  - One process takes place of another
  - This “switch” is obviously not going to happen using calling convention, as no “call” is happening
  - Code for context switch must be in assembly!

NEXT: XV6 code overview

1. Understanding how traps are handled
2. How timer interrupt goes to scheduler
3. How scheduling takes place
4. How a “blocking” system call (e.g. read()) “blocks”