Process control block (PCB)

```
struct process {
    int pid;
    unsigned base;
    unsigned limit;
    struct fd *fdtable;
    struct list threads;
};
```

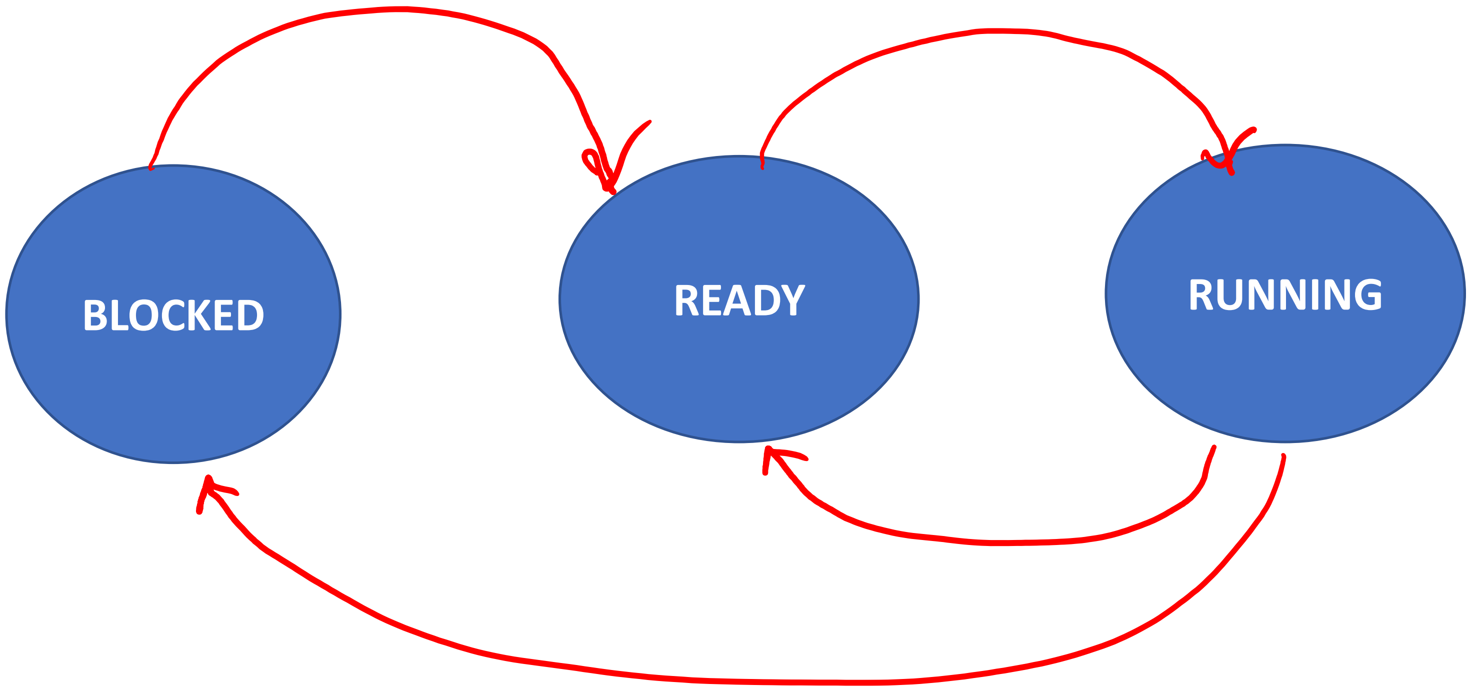
Thread control block (TCB)

```
struct thread {
    int tid;
    struct process *pcb;
    void *esp;
    unsigned regs[8];        // general purpose registers
    int state;
};
```

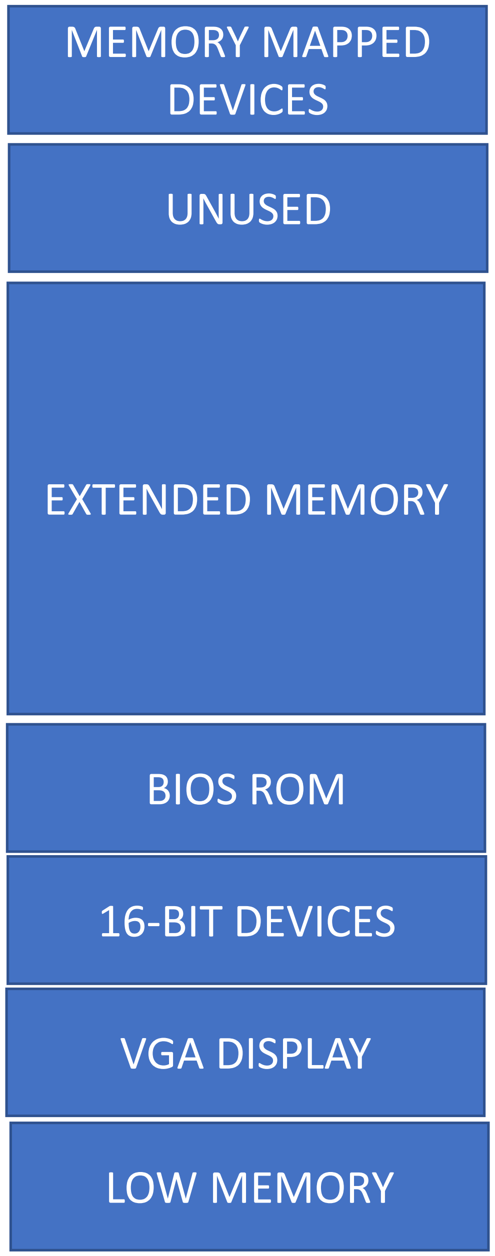
Threads

- thread state can be one of these three states
  - RUNNING, READY, BLOCKED

Threads



Memory map

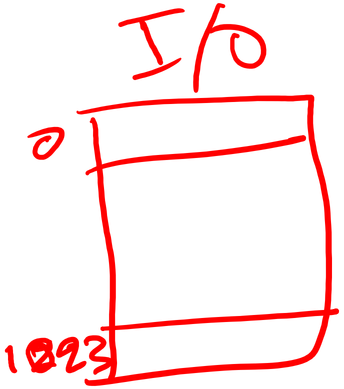

MEMORY MAPPED DEVICES
UNUSED
EXTENDED MEMORY
BIOS ROM
16-BIT DEVICES
VGA DISPLAY
LOW MEMORY

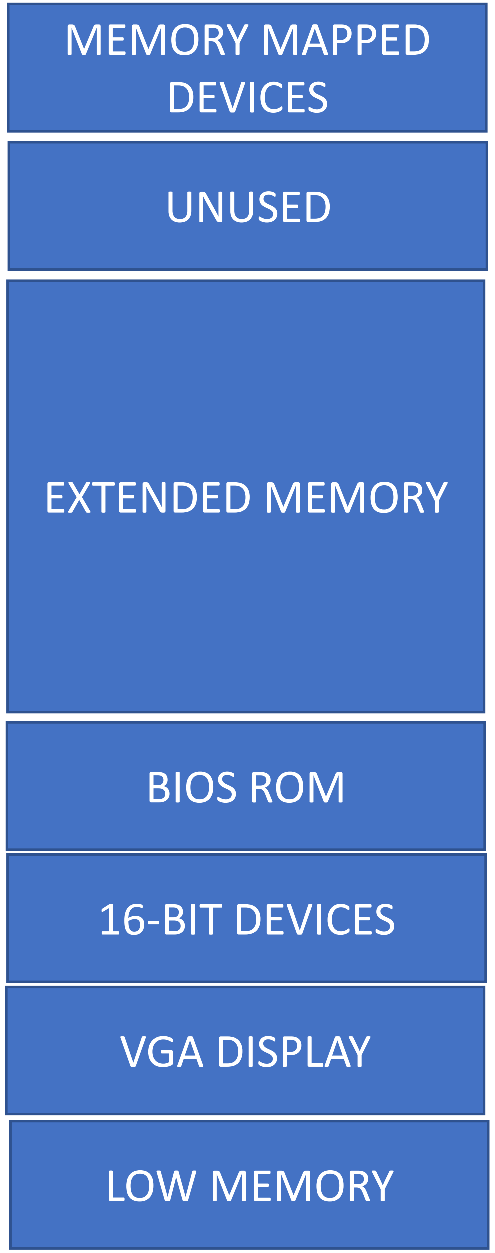
Port I/O

- Use dedicated I/O space
  - Only 1024 I/O address
  - accessed using in/out instructions

in $100

out $100





Line printer (Port I/O)

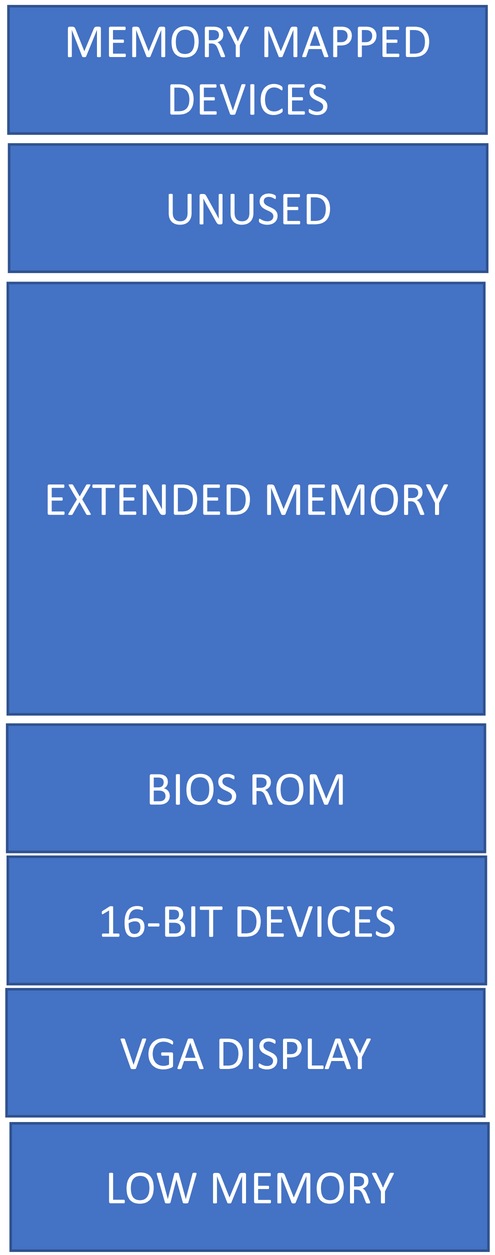
```
#define DATA_PORT 0x378
#define STATUS_PORT 0x379
#define BUSY 0x80
#define CONTROL_PORT 0x37A
#define STROBE 0x01

void lpt_putc(int c) {
  while((inb(STATUS_PORT) & BUSY) == 0) ; /* wait for printer to consume previous byte */
  outb(DATA_PORT, 'c'); /* put the byte on the parallel lines */
  outb(CONTROL_PORT, STROBE); /* tell the printer to look at the data */
  outb(CONTROL_PORT, 0);
}
```

Memory mapped I/O

- Some addresses in the physical address space are reserved for devices

- Software talks to the devices by reading/writing to these addresses



Line printer (MMIO)

```
volatile int *DATA_REG = 0xffffff00;
volatile int *STATUS_REG  = 0xffffff04;
volatile int* CONTROL_REG = 0xffffff0C;
#define BUSY  0x80
#define STROBE 0x01

void lpt_putc(int c) {
  while ((STATUS_REG[0] & BUSY) == 0) ; /* wait for printer to consume previous byte */
  DATA_REG[0] = 'c'; /* put the byte on the parallel lines */
  CONTROL_REG[0] = STROBE; /* tell the printer to look at the data */
  CONTROL_~~PORT~~REG[0] = 0;
}
```

Volatile vs non-volatile

```
while ((STATUS_REG[0] & BUSY) == 0) ;
```

```
int reg = STATUS_REG[0];
while ((reg & BUSY) == 0);
```

Direct memory access (DMA)

```
#define TX_STATUS_PORT   0x400
#define TX_ADDR_PORT     0x401
#define RX_STATUS_PORT   0x402
#define RX_ADDR_PORT     0x403

tx_packet (char *buf) {                  // transmit buf to the network
  outl (TX_STATUS_PORT, 0);
  outl (TX_ADDR_PORT, virt_to_phys(buf));
  while (inl(TX_STATUS_PORT) == 0);   // polling until buf is sent
}
```

waiting

Direct memory access

```
#define TX_STATUS_PORT   0x400
#define TX_ADDR_PORT      0x401
#define RX_STATUS_PORT   0x402
#define RX_ADDR_PORT      0x403

rx_packet (char *buf) {               // receive network packet in buf
   outl (RX_STATUS_PORT, 0);
   outl (RX_ADDR_PORT, virt_to_phys(buf));
   while (inl(RX_STATUS_PORT) == 0);   // wait until data is received in buf
}
```

Polling

- Periodically checking for the occurrence of an event
  - e.g. after every 100 ms network device checks for the arrival of a packet

- Polling is not useful in all cases
  - Specifically, when the device events are not frequent, e.g.,
    - the keyboard input
    - low throughput network channel

Interrupts

```
#define TX_STATUS_PORT   0x400
#define TX_ADDR_PORT     0x401
#define RX_STATUS_PORT   0x402
#define RX_ADDR_PORT     0x403

rx_packet (char *buf) {
  outl (RX_STATUS_PORT, 0);
  outl (RX_ADDR_PORT, virt_to_phys(buf));
  //while (inl(RX_STATUS_PORT) == 0);
}
```

```
/* called after packet is received */
rx_intr () {
  addr_t phys = inl(RX_ADDR_PORT);
  char *buf = phys_to_virt (phys);
  return buf;
}
/* called after every 100 ms */
rx_poll() {
  addr_t phys = inl(RX_ADDR_PORT);
  char *buf = phys_to_virt (phys);
  return buf;
}
```

Disk driver

- Read disk driver section from chapter 3 of xv6-book for a real device driver

Shell

- What if the shell doesn't call wait system call
- The shell can execute another command even if the previous command hasn't finished yet

```
while (1) {
    write (1, "$ ", 2);
    readcommand (0, command, args);
    if ((pid = fork ()) == 0) {
        exec (command, args, 0);
    } else if (pid > 0) {
        status = wait (0);
    } else
        printf ("Failed to fork\n");
}
```

Background process in shell

- ls &
  - run “ls” in background
  - shell doesn’t wait for ls to complete before reading another command

```
while (1) {
    write (1, “$ “, 2);
    readcommand (0, command, args);
    if ((pid = fork ()) == 0) {
        exec (command, args, 0);
    } else if (pid > 0) {
        //status = wait (0);
    } else
        printf (“Failed to fork\n”);
}
```

Problem with not calling wait

- The child process is not completely destroyed until the parent process calls the wait system call

- The wait system call returns the exit status of the child process

- The child process becomes a zombie process after doing an exit system call until the parent process does the wait system call

Zombie process

- A zombie process is a resource leak
- The shell somehow need to call the wait system call for the background process

Signals

- SIGINT (ctrl + c)
- SIGSTOP (ctrl + z)
- SIGCHLD (child process terminates)
- SIGTRAP (processor executed int3)
- SIGSEGV (memory access outside limit)
- and may more

Signal system call

```
sighandler_t signal(int signum, sighandler_t handler);
```

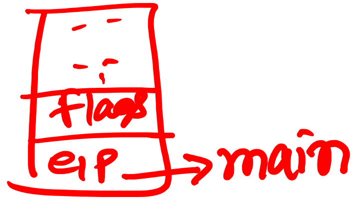
- Registers a handler for a given signal with the OS. The handler is called when the signal is generated.
- Signals invoke an asynchronous method in the user address space, similar to interrupts.

Signals

```
/* sigHandler is called every time "ctrl + c" is pressed */
void sigHandler(int signal) {
   printf ("received %d signal\n", signal);
}

main () {
   signal (SIGINT, sigHandler);
   ...
}
```

user process



terminal_driver()
{
SIGINT
sigHandler
}

kill system call

- int kill (pid_t pid, int sig)
- Send a signal to another process
- e.g., a process can send SIGKILL signal to kill a target process

Background process in shell

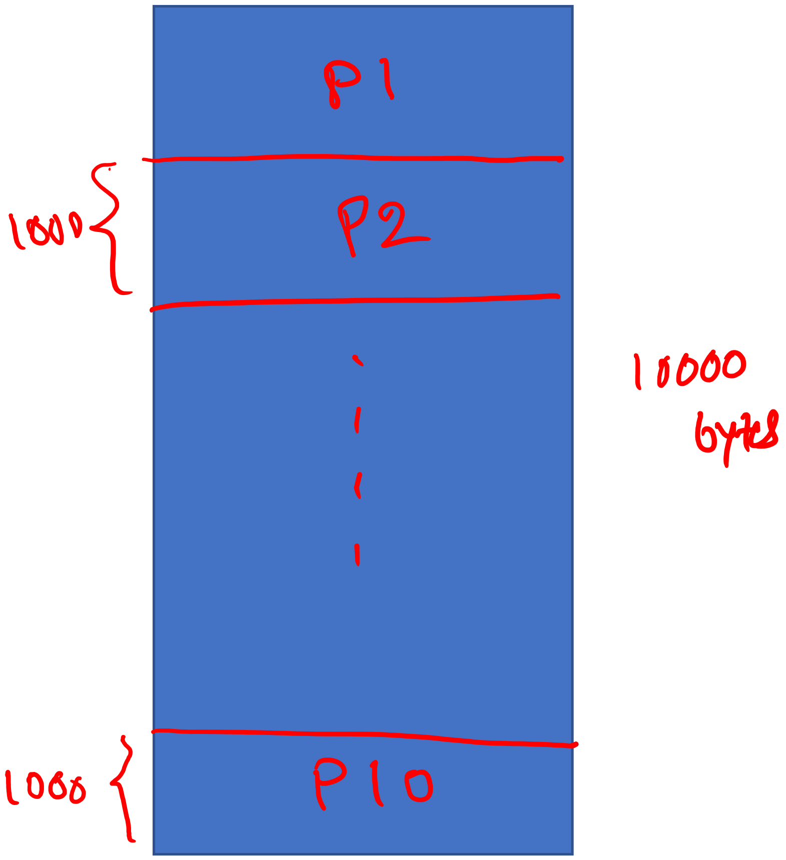
```
childExit()
{
    wait (0);
}
```

```
signal (SIGCHLD, childExit);
while (1) {
    write (1, "$ ", 2);
    readcommand (0, command, args);
    if ((pid = fork ()) == 0) {
        exec (command, args, 0);
    } else if (pid > 0) {
        //status = wait (0);
    } else
        printf ("Failed to fork\n");
}
```

Fragmentation

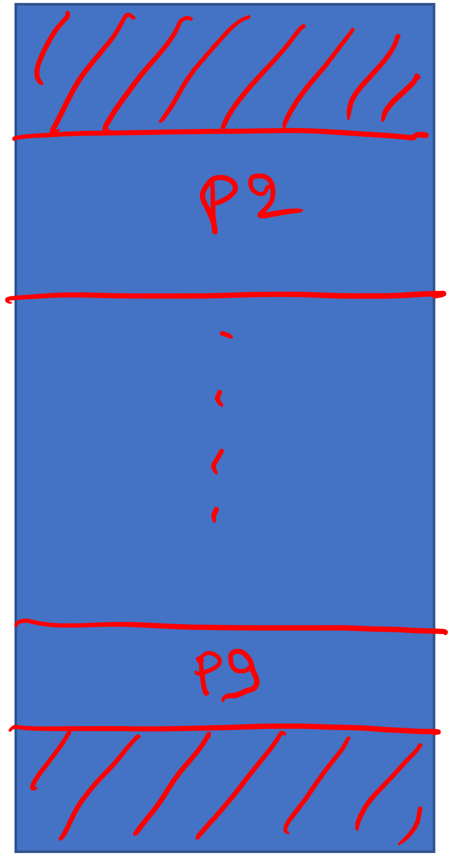
- Suppose the RAM of size 10000 bytes is mapped at location 0 – 10000

- And we have 10 processes (P1 - P10) each of size 1000 bytes



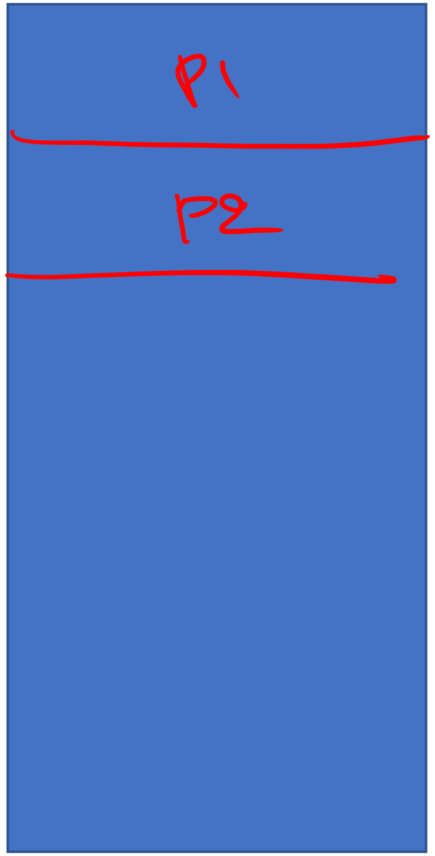
Fragmentation

- Let's say P1 and P10 have finished their execution
- At this point, the OS wants to load a new process P11 of size 2000
- The OS needs to relocate either P2 or P9 to accommodate the new process
  - need to copy the entire process



Fragmentation

- Similarly, if a process needs more RAM during its execution and consecutive addresses are not available the OS may need to relocate the entire process



Process address space in segmentation

- The process address space is limited by the amount of RAM available on the system

Paging

- Due to the limitations of segmentation the x86 hardware has an additional memory management hardware called paging hardware

- In paging scheme, the process virtual address space is ($0 - 2^{32}$-1) for every process