Processes

A process is a program in execution. Processes are identified by a **process identifier** (pid). It is composed of multiple parts:

- Text section: the program code
- Data section: contains global variables (initialized and uninitialized)
- Heap: memory dynamically allocated during runtime
- Stack: contains temporarily variables, such as function parameters, return addresses, local variables

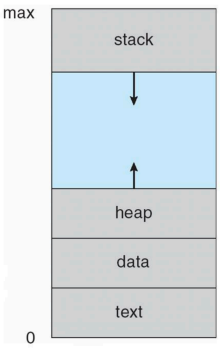

Figure 1: Memory layout for a process

A process during execution cycles through the following states:

- new: the process is created
- ready: The process is in a queue and is waiting to be assigned to a processor
- running: Instructions are being executed
- waiting: The process is in a queue and is waiting for some event to occur (ex. a memory transfer, an I/O)
- terminated: The process has finished execution

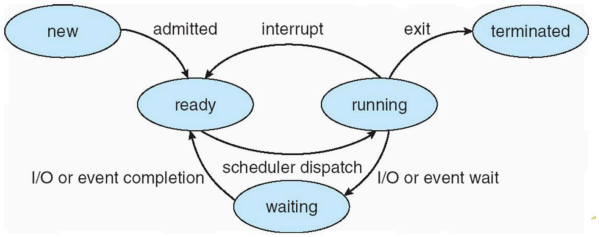

Figure 2: Process state machine

The information about the state of the process is stored in the RAM in the process control block (PCB). It contains the following information:

- Process state: running, waiting, etc.
- Program counter: location of next instruction
- CPU registers: contents of registers used by the process
- CPU scheduling information: priorities, scheduling queue pointers
- Memory-management information: memory allocated to the process
- Accounting/Debug information: CPU used, clock time elapsed since start, time limits
- I/O status information: I/O devices allocated to process, list of open files

In Linux the PCB for every process is stored as a file in the /proc folder: `less /proc/<pid::self>/status .`

When a process is stopped it saves its state in the PCB and if reloads it when it resumes executing. The time when the CPU stores the PCB of a process and loads the PCB of another process is called **context switch**. Context switches can be categorized in:

- voluntary c. s.: the process stops itself because needs to wait for a resource
- nonvoluntary c. s.: the processor decides to switch process

Multithreading

A process can execute multiple instructions at once by using multiple threads. Each thread has its own program counter and uses different registers, therefore all this information has to be also stored in the PCB.

Scheduling

The CPU has a process scheduler, which decides which process to execute. The scheduler stores the processes in various queues:

- Job queue: set of all processes in the system
- Ready queue: set of all processes residing in main memory, ready and waiting to execute
- Device queues: set of processes waiting for an I/O device

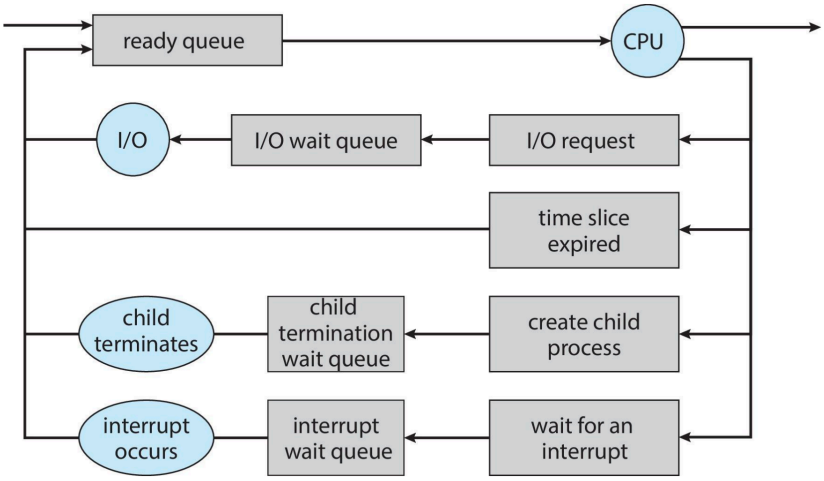

Figure 3: Process queues

Process creation

A process can create other *child* processes, which in turn can have other children. Therefore processes are arranged in a tree data structure. In Linux the process tree can be printed using `pstree`.

The parent and children have different options for sharing resources:

- Parent and children share all resources
- Children share subset of parent's resources
- Parent and child share no resources

Moreover they have different options for execution:

- Parent and children execute concurrently
- Parent waits until children terminate

In a Linux system the root process that spawns all other processes is called *systemd*. In the UNIX processes are managed using the following system calls:

- `fork()`: creates a new process

- `exec()`: replaces the parent's memory with the children's one (machine code, data, heap, and stack)
- `wait()`: called by parent to wait for the end of the child's execution

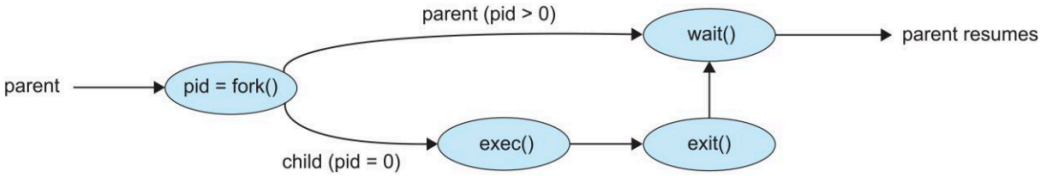

Figure 4: Creation of children processes

```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t pid;

    // Returns 0 if called from the child process
    // Returns the PID of the child process of -1 on error
    // if called from the parent process
    pid = fork();

    if (pid < 0) {
        fprintf(stderr, "Fork failed\n");
        return 1;
    } else if (pid == 0) {
        printf("Child print\n");
    } else {
        wait(NULL); // Waits for the child process to finish executing
        printf("Parent print after child\n");
    }
}
```

Listing 1: A process that spawns a child process and waits for its termination

Communication between processes

Processes can communicate using:

- shared memory: processes that wish to communicate create a shared area of memory, that is managed directly by the processes
- message passing

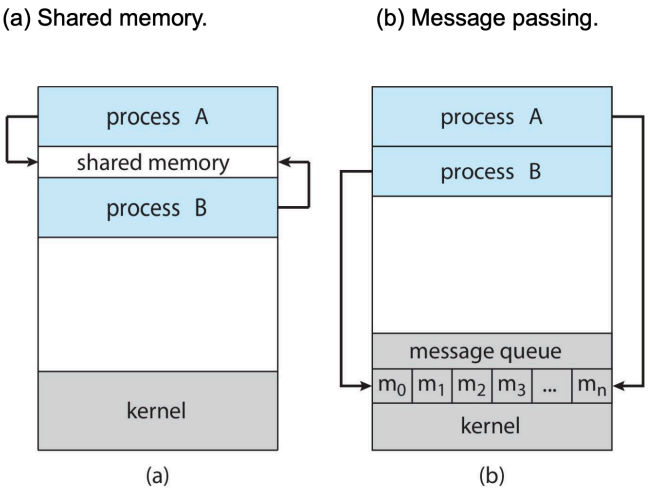

Figure 5: Models of communication between processes

Shared memory

Processes can communicate using shared memory by creating a dedicated area in memory and writing and reading to it. To access shared memory it processes need to map it to memory using memory mapping. Shared memory can be accessed by their name.

Memory-mapped files

In modern operating systems files, shared memory objects and other resources that can be addressed by a file descriptor can be accessed by processes using a procedure called "memory mapping". When memory mapping is used, a byte-to-byte correlation with a part of memory and the resource is established. This means that the processor can access the file very quickly, as if it was stored in memory. In UNIX this feature is provided by the `mmap()` system call.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <sys/types.h>

int main()
{
  const int SIZE = 16;
  /* name of the shared memory */
  const char *name = "OS";
  const char *message0 = "Hello world ";
  const char *message1 = "I'm a shared message";

  /* shared memory file descriptor */
  int shm_fd;
  /* pointer to a shared memory object */
  void *ptr;

  /* create the shared memory segment */
  shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

  /* configure the size of the shared memory segment */
```

```
  ftruncate(shm_fd,SIZE);

  /* now map the shared memory segment in the address space of the process */
  ptr = mmap(0,SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
  if (ptr == MAP_FAILED) {
    printf("Map failed\n");
    return -1;
  }

  /**
   * Now write to the shared memory region. Note we must increment the value of ptr
after each write.
   */
  sprintf(ptr,"%s",message0);
  ptr += strlen(message0);
  sprintf(ptr,"%s",message1);
  ptr += strlen(message1);
  return 0;
}
```

Listing 2: A process that creates a shared memory area and writes to it

```
 #include <stdio.h>
 #include <stdlib.h>
 #include <unistd.h>
 #include <fcntl.h>
 #include <sys/shm.h>
 #include <sys/stat.h>
 #include <sys/mman.h>

 int main()
 {
   const char *name = "OS";
   const int SIZE = 16;

   int shm_fd;
   void *ptr;
   int i;

   /* open the shared memory segment */
   shm_fd = shm_open(name, O_RDONLY, 0666);
   if (shm_fd == -1) {
     printf("shared memory failed\n");
     exit(-1);
   }

   /* now map the shared memory segment in the address space of the process */
   ptr = mmap(0,SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);
   if (ptr == MAP_FAILED) {
     printf("Map failed\n");
     exit(-1);
   }

   /* now read from the shared memory region */
   printf("%s",(char *)ptr);

   /* remove the shared memory segment */
   if (shm_unlink(name) == -1) {
```

```
    printf("Error removing %s\n",name);
    exit(-1);
  }

  return 0;
}
```

Listing 3: A process that opens a shared memory area and reads from it

Message passing

Processes can communicate without using shared memory by using message passing. This can be physically implemented in the following ways:

- Shared memory (we already saw that)
- Hardware bus
- Network

We can distinguish the channel on a logical level in the following ways: Direct or indirect

- Synchronous or asynchronous
- Automatic or explicit buffering

Pipes

Pipes provide a way for to processes to communicate directly with each other. We can distinguish among two different types of pipes: ordinary pipes and named pipes. Pipes are accessed using the file descriptor.

Ordinary pipes

Ordinary pipes cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created. Ordinary pipes are unidirectional, meaning that the parent process can only write to it and the child process can only read from it. In Windows they are called ordinary pipes.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>

#define BUFFER_SIZE 25
#define READ_END  0
#define WRITE_END 1

int main(void)
{
  char write_msg[BUFFER_SIZE] = "Greetings";
  char read_msg[BUFFER_SIZE];
  pid_t pid;
  int fd[2];

  /* create the pipe */
  if (pipe(fd) == -1) {
    fprintf(stderr,"Pipe failed");
    return 1;
  }

  /* now fork a child process */
  pid = fork();
```

```
    if (pid < 0) {
      fprintf(stderr, "Fork failed");
      return 1;
    }

    if (pid > 0) {  /* parent process */
      /* close the unused end of the pipe */
      close(fd[READ_END]);

      /* write to the pipe */
      write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

      /* close the write end of the pipe */
      close(fd[WRITE_END]);
    }
    else { /* child process */
      /* close the unused end of the pipe */
      close(fd[WRITE_END]);

      /* read from the pipe */
      read(fd[READ_END], read_msg, BUFFER_SIZE);
      printf("child read %s\n",read_msg);

      /* close the write end of the pipe */
      close(fd[READ_END]);
    }

    return 0;
  }
```

Listing 4: A process that spawn a child and communicates to it using an ordinary pipe

Named pipes

Named pipes can be accessed without a parent-child relationship. They are bidirectional and multiple processes can read and write to it. When no process holds a reference to the file descriptor the pipe is destroyed by the system.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

#define BUFFSIZE 512
#define err(mess) { fprintf(stderr,"Error: %s.", mess); exit(1); }

void main()
{
    int fd, n;
    char buf[BUFFSIZE];
    mkfifo("fifo_x", 0666);
    if ( (fd = open("fifo_x", O_WRONLY)) < 0)
        err("open")
    while( (n = read(STDIN_FILENO, buf, BUFFSIZE) ) > 0) {
        if ( write(fd, buf, n) != n) {
```

```
                err("write");
            }
        }
        close(fd);
}
```

Listing 5: A process that creates a named pipe and writes the content from the standard input in the pipe

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

#define BUFFSIZE 512
#define err(mess) { fprintf(stderr,"Error: %s.", mess); exit(1); }

void main()
{
    int fd, n;
    char buf[BUFFSIZE];
    mkfifo("fifo_x", 0666);
    if ( (fd = open("fifo_x", O_WRONLY)) < 0)
        err("open")
    while( (n = read(STDIN_FILENO, buf, BUFFSIZE) ) > 0) {
        if ( write(fd, buf, n) != n) {
            err("write");
        }
    }
    close(fd);
}
```

Listing 6: A process that opens a named pipe and prints the content in the pipe to the standard output