



Processes - Part I

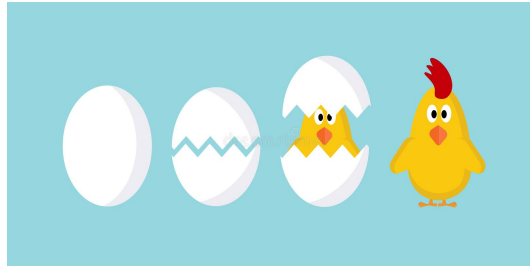
Amir H. Payberah
payberah@kth.se
Nov. 1, 2023



What Is A Process?

Process

An **instance** of a **program** running.



<https://tinyurl.com/53pecc99>

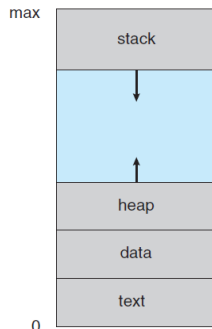
Program vs. Process

- ▶ **Program** is a **passive** entity stored on disk (**executable file**).
- ▶ **Process** is an **active** entity.
- ▶ **Program** becomes **process** when executable file loaded into **memory**.
- ▶ One program can be several processes.

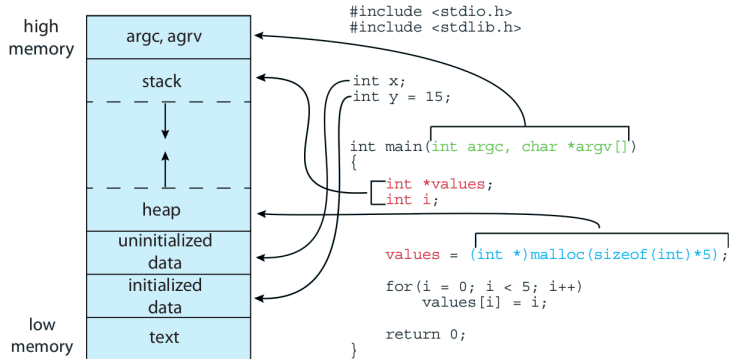


Parts of a Process (1/2)

- ▶ A process is more than the program code.
- ▶ Multiple parts of a process:
 - Text section: the executable code
 - Data section: global variables
 - Heap section: memory that is dynamically allocated during program run time
 - Stack section: temporary data storage when invoking functions (e.g., function parameters, return addresses, and local variables)



Parts of a Process (2/2)



Process Control Block (PCB)

- ▶ The **information** of each process.

process state
process number
program counter
registers
memory limits
list of open files
...



Process ID

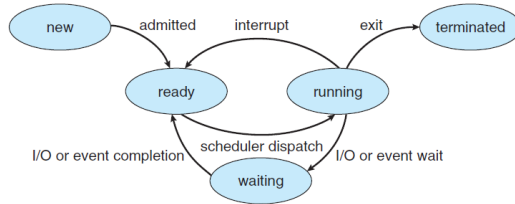
- ▶ Each process is assigned a **unique identifier**, the **process ID (PID)**.
- ▶ The **kernel** allocates PIDs to processes in a strictly **linear** fashion.
- ▶ The **getpid()** system call **returns the PID** of the invoking process.

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

pid_t getpid(void);
```


Process States

- ▶ As a process **executes**, it changes **state**.



- ▶ **new**: The process is being **created**.
- ▶ **ready**: The process is **waiting** to be assigned to a processor.
- ▶ **running**: Instructions are being **executed**.
- ▶ **waiting**: The process is **waiting** for some **event** to occur.
- ▶ **terminated**: The process has **finished** execution.

Process Scheduling

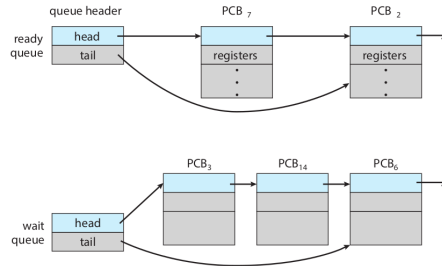


Process Scheduling

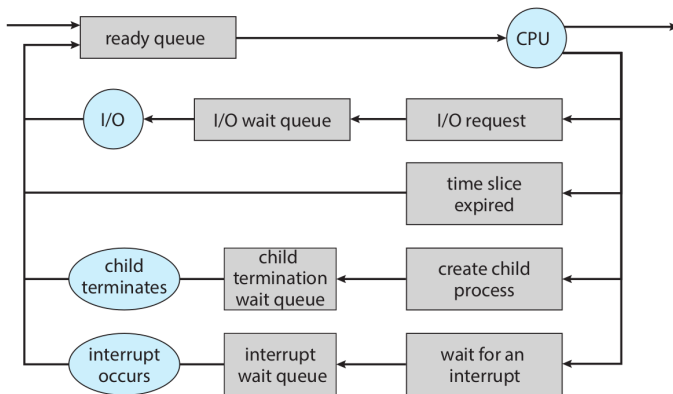
- ▶ **Process scheduler** selects among **available processes** for next execution on CPU core.
- ▶ **Goal:** **Maximize CPU use, quickly switch processes** onto CPU core

Scheduling Queues

- ▶ **Ready queue:** set of **all processes** residing in **main memory**, ready and waiting to execute.
- ▶ **Wait queues:** set of processes **waiting** for an **event** (e.g., I/O device).

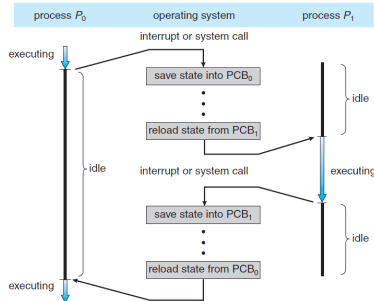


Queuing Diagram



Context Switching

- ▶ When CPU **switches** to another process:
 - The **state** of the **old process** is **saved** by the system.
 - The **saved state** of the **new process** is **loaded** via a context switch.
 - Called **context switching**.



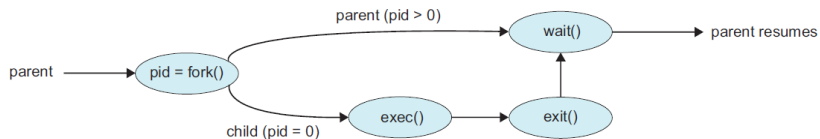
- ▶ Context of a process represented in the **PCB**.

Operations on Processes

Operations on Processes

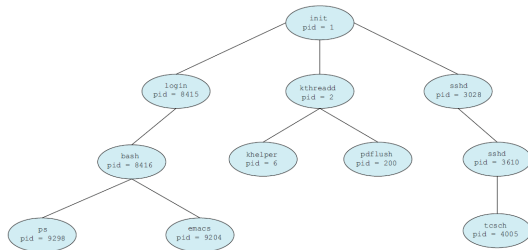
► OS must provide mechanisms for:

- Process creation
- Process termination



Process Creation (1/5)

- ▶ A process may **create** several **new processes**.
 - The **creating process**: the **parent** process.
 - The **new processes**: the **children** processes.
- ▶ These processes are forming a **tree of processes**.



it lists complete information for all active processes in the system
`ps -el`

Process Creation (2/5)

- ▶ `fork()` creates a new process.
- ▶ The new process (child) running the same image as the current one (parent).
- ▶ `fork()` is called once, but it returns twice.
 - The PID of the new child → to the parent.
 - 0 → to the child.

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

pid_t fork(void);
```

Process Creation (3/5)

```
pid_t pid = fork();

if (pid == -1) {
    perror("fork");
    exit(1);
}

if (pid > 0)
    printf("I am the parent of pid = %d!\n", pid);
else
    printf("I am the child!\n");
```

Process Creation (4/5)

- ▶ `exec()` **executs** a **new program**.
- ▶ Used after `fork()` to replace the process' memory space with a new program.

```
#include <unistd.h>
int execl(const char *path, const char *arg, ...);
```

Process Creation (5/5)

```
pid_t pid = fork();

if (pid == -1) {
    perror("fork");
    exit(1);
}

if (pid == 0) { // the child
    const char *args[] = {"windlass", NULL};
    int ret;
    ret = execv("/bin/windlass", args);
    if (ret == -1) {
        perror("execv");
        exit(1);
    }
}
```



Process Termination (1/4)

- ▶ Process executes **last statement** and then asks the OS to delete it.
- ▶ Returns **status data** from the **child** to the **parent**.
- ▶ Process **resources** are **deallocated** by the OS.

Process Termination (2/4)

- ▶ The `exit()` then instructs the kernel to **terminate** the process.
- ▶ The `status` is used to denote the process's **exit status**.

```
#include <stdlib.h>  
  
void exit(int status);
```



Process Termination (3/4)

- ▶ The **parent** process may **wait** for **termination** of a **child** via **wait()**.
- ▶ The **wait()** returns the **status information** and the **PID** of the **terminated process**.
- ▶ If a process has terminated, but whose **parent** has not yet called **wait()**, the process is a **zombie**.
- ▶ If the parent terminated **without invoking wait()**, the process is an **orphan**.
 - In Linux, the **init** process becomes the parent of all orphans.

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
```

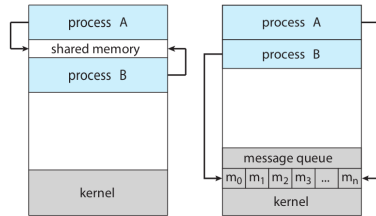

Process Termination (4/4)

```
int main (void) {  
    int status;  
    pid_t pid;  
  
    if (fork() == 0) return 1; // the child  
  
    pid = wait(&status);  
  
    if (pid == -1) perror("wait");  
  
    printf("pid = %d\n", pid);  
  
    return 0;  
}
```

Inter-Process Communication (IPC)

Inter-Process Communication (IPC)

- ▶ **IPC** mechanisms allow processes to **exchange** data.
- ▶ **Two** models of IPC
 - Shared memory
 - Message passing



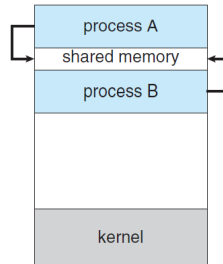
Shared-memory

Message-passing

Shared Memory

Shared Memory (1/4)

- ▶ An **area of memory shared** among the **processes** that wish to communicate.
- ▶ It resides in the **address space** of the **process creating the shared-memory segment**.



Shared Memory (2/4)

- ▶ `shm_open()` creates and opens a new shared memory object or opens an existing object.
- ▶ `mmap()` creates a new mapping in the virtual address space of the calling process.
- ▶ `shm_unlink()` removes a shared memory object.

```
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/mman.h>

int shm_open(const char *name, int oflag, mode_t mode);

void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);

int shm_unlink(const char *name);
```

Shared Memory (3/4)

► Producer

```
int SIZE = 4096;
char *my_shm = "/tmp/myshm";
char *write_msg = "hello";
char *addr;
int fd;

// create the shared memory object
fd = shm_open(my_shm, O_CREATE | O_RDWR, 0666);

// configure the size of the shared memory object
ftruncate(fd, SIZE);

// memory map to the shared memory object
addr = mmap(NULL, SIZE, PROT_WRITE, MAP_SHARED, fd, 0);

// write to the shared object
sprintf(addr, "%s", write_msg);
```

Shared Memory (4/4)

► Consumer

```
int SIZE = 4096;
char *my_shm = "/tmp/myshm";
char *addr;
int fd;

// open the shared memory object
fd = shm_open(my_shm, O_RDONLY, 0666);

// memory map to the shared memory object
addr = mmap(NULL, SIZE, PROT_READ, MAP_SHARED, fd, 0);

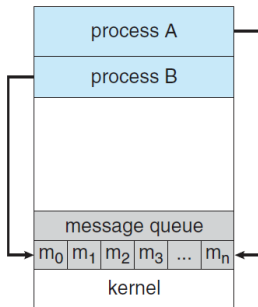
// read from to the shared object
printf("%s", (char *)addr);

// remove the shared memory object
shm_unlink("my_shm");
```


Message Passing

Message Passing

- ▶ Communicating with each other **without resorting to shared variables**.
- ▶ Useful in a **distributed environment**: processes on different computers.





Message Passing IPC Facilities

- ▶ Pipe
- ▶ FIFO (named pipe)
- ▶ Message queue

Pipe



Pipe (1/4)

- ▶ Pipes are **unidirectional**, allowing only **one-way** communication.
- ▶ Require **parent-child** relationship between communicating processes.

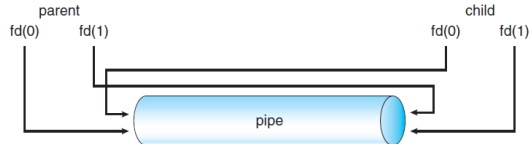
```
ls | wc -l
```

Pipe (2/4)

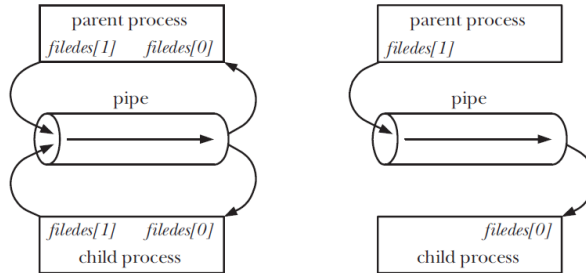
- ▶ `pipe()` creates a **new pipe**.
- ▶ It returns **two** open file descriptors in `fd`:
 - `fd[0]` to **read** from the pipe
 - `fd[1]` to **write** to the pipe

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

```
int pipe(int fd[2]);
```



Pipe (3/4)



[Michael Kerrisk, The Linux Programming Interface, No Starch Press, 2010]

Pipe (4/4)

```
int BUFFER_SIZE = 25;
char write_msg[BUFFER_SIZE] = "hello";
char read_msg[BUFFER_SIZE];
int fd[2];

pipe(fd); // Create the pipe

switch (fork()) {
    case -1: // fork error
        break;
    case 0: // Child
        close(fd[1]); // Close unused write end
        read(fd[0], read_msg, BUFFER_SIZE);
        printf("read %s", read_msg);
        break;
    default: // Parent
        close(fd[0]) // Close unused read end
        write(fd[1], write_msg, strlen(write_msg) + 1);
        break;
}
```


FIFO



FIFO (1/4)

- ▶ **FIFO** is similar to a pipe, but it has a **name** within the file system and is opened in the same way as a **regular file**.
- ▶ Communication is **bidirectional**.
- ▶ **No parent-child** relationship is necessary.
- ▶ **Several processes** can use a FIFO for communication.

FIFO (2/4)

- ▶ The `mkfifo()` function creates a new FIFO.

```
#include <sys/stat.h>

int mkfifo(const char *pathname, mode_t mode);
```

FIFO (3/4)

► Producer

```
char *my_fifo = "/tmp/myfifo";  
char *write_msg = "hello";  
int fd;  
  
// Create the FIFO (named pipe)  
mkfifo(my_fifo, 0666);  
  
// Write "hello" to the FIFO  
fd = open(my_fifo, O_WRONLY);  
write(fd, write_msg, strlen(write_msg));  
close(fd);  
  
// Remove the FIFO  
unlink(my_fifo);
```

FIFO (4/4)

► Consumer

```
int MAX_SIZE = 100;
char *my_fifo = "/tmp/myfifo";
char buf[MAX_SIZE];
int fd;

// Open the FIFO
fd = open(my_fifo, O_RDONLY);

// Read the message from the FIFO
read(fd, buf, MAX_SIZE);
printf("Received: %s\n", buf);

// Close the FIFO
close(fd);
```

Message Queue



Message Queue (1/6)

- ▶ **Message queues** allows processes to exchange data in the form of **messages**.
- ▶ In message queue the **consumer receives whole messages**, as written by the producer.
 - It is **not possible** to read part of a message.

Message Queue (2/6)

- ▶ `mq_open()` creates a new message queue or opens an existing queue.
- ▶ `mq_close()` closes the message queue descriptor `mqdes`.
- ▶ `mq_unlink()` removes the message queue identified by `name`.

```
#include <fcntl.h>
#include <sys/stat.h>
#include <mqueue.h>

mqd_t mq_open(const char *name, int oflag, ...);

int mq_close(mqd_t mqdes);

int mq_unlink(const char *name);
```


Message Queue (3/6)

- ▶ `mq_send()` adds the message `msg_ptr` to the message queue.
- ▶ `mq_receive()` removes the oldest message from the message queue.

```
#include <mqqueue.h>

int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned int msg_prio);

ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned int *msg_prio);
```

Message Queue (4/6)

- Specifies **attributes** of a message queue.

```
struct mq_attr {  
    long mq_flags; // Message queue description flags  
    long mq_maxmsg; // Maximum number of messages on queue  
    long mq_msgsize; // Maximum message size (in bytes)  
    long mq_curmsgs; // Number of messages currently in queue  
};
```

Message Queue (5/6)

► Producer

```
char *my_mq = "/mymq";  
char *write_msg = "hello";  
mqd_t mqd;  
  
// Open an existing message queue  
mqd = mq_open(my_mq, O_WRONLY);  
  
// Write "hello" to the message queue  
mq_send(mqd, write_msg, strlen(write_msg), 0);  
  
// Close the message queue  
mq_close(mqd);
```

Message Queue (6/6)

► Consumer

```
int MAX_SIZE = 100;
int MAX_NUM_MSG = 10;
char *my_mq = "/mymq";
char buf[MAX_SIZE];
mqd_t mqd;
struct mq_attr attr;

// Form the queue attributes
attr.mq_maxmsg = MAX_NUM_MSG;
attr.mq_msgsize = MAX_SIZE;

// Create message queue
mqd = mq_open(my_mq, O_RDONLY | O_CREAT, MQ_MODE, &attr);

// Read the message from the message queue
mq_receive(mqd, buf, MAX_NUM_MSG, NULL);
printf("Message: %s\n", buf);

// Close the message queue
mq_close(mqd);
```

Summary

Summary

- ▶ **Process** vs. **Program**
- ▶ **Process Control Block (PCB)**
- ▶ **Process states**: new, running, waiting, ready, terminated
- ▶ **Process scheduling**: scheduling queues, context switching
- ▶ **Process operations**: creation (parent-child), termination
- ▶ **Inter-Process Communication**: shared memory vs. message passing
- ▶ **Message passing**: pipe, FIFO, message queue

Questions?