

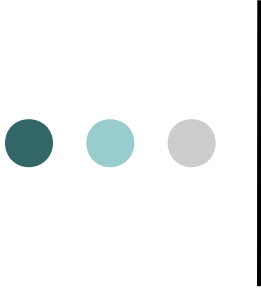
Processos Unix (perspectiva do programador)



"*Race conditions*"

- Verificam-se quando o resultado final das operações de um conjunto de processos/tarefas depende de condições não controláveis.
 - Essas condições podem afectar ordem de execução das respectivas instruções, originando diferentes resultados.
 - Exemplos de condições não controláveis: carga do sistema, sequência de *inputs* para a aplicação.
- Exemplo: Qual a ordem das impressões no programa abaixo?

```
int main() {  
    fork();  
    printf("%d\n",getpid());  
}
```



Race conditions (versão 2 do exemplo)

- Pretende-se que a ordem de impressão seja: 1º pai; 2º filho.

```
int main() {  
    int pid = fork();  
    if(pid==0)  
        sleep(1); //será que 1 segundo chega?  
  
    printf("%d\n",getpid());  
}
```

- E interferências de outros processos do sistema?
 - Solução pobre e sujeita a problemas.



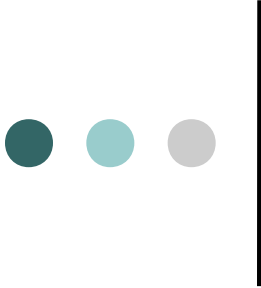
Análise do exemplo anterior: equivalência a dois programas

```
//processo "pai"
int main() {
    pid = PID_DO_FILHO;
    if(pid==0)
        sleep(1);

    printf("%d\n",
           getpid());
}
```

```
//processo "filho"
int main() {
    pid = 0;
    if(pid==0)
        sleep(1);

    printf("%d\n",
           getpid());
}
```



Versão 3 do exemplo de "*race conditions*": solução com sinais

```
void sig_usr(int signo)
{ sigflag = 1; }

int main() {
    int pid;
    sigset_t newmask, zeromask;

    signal(SIGUSR1, sig_usr)

    sigemptyset(&zeromask);
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGUSR1);
    sigprocmask(SIG_BLOCK, &newmask, NULL)

    //...
```



(cont.)

```
//...
```

```
pid = fork();
if(pid==0){
    while (sigflag == 0)
        sigsuspend(&zeromask); /* wait for parent */

    printf("%d\n",getpid());
} else {
    printf("%d\n",getpid());
    kill(pid, SIGUSR1);
}
}
```

- A utilização do sigflag não deveria tornar o bloqueio de sinais desnecessário?
- Dica: o que acontece se o sinal for gerado entre o while() e o sigsuspend()? ->
- Porquê o while, com o uso da flag? -> outros sinais
- Porquê não usar o pause em vez de sigsuspend()?



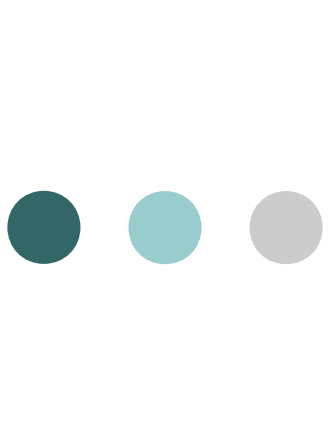
Atomicidade!

- `sigsuspend(&zeromask);`

≠

- `sigprocmask(SIG_SETMASK, &zeromask, NULL);`
`pause();`

- Se o sinal for gerado entre a execução do *sigprocmask()* e do *pause()* o processo vai perder esse sinal e ficar bloqueado até receber novo sinal.



Comunicação entre processos: pipes.



Pipes

- Mecanismo de comunicação de dados entre processos
- Pode ser visto como um canal de comunicação (“tubo”) unidirecional
 - Transferência sequencial de bytes
 - *First in First out* (FIFO)



Pipes

- Extremidade de envio é tratada como um ficheiro aberto para escrita
 - Envios são feitos com funções de escrita para ficheiros.
- Extremidade de receção é tratada como um ficheiro aberto para leitura
 - Receção dos dados é feita com funções de leitura de ficheiros.
 - Cada byte só pode ser lido uma vez (transmissão sequencial)
- Permite comunicações *half-duplex*:
 - Ambas as extremidades do canal podem ser abertas pelos diferentes processos.



Pipes: anónimos ou com nome

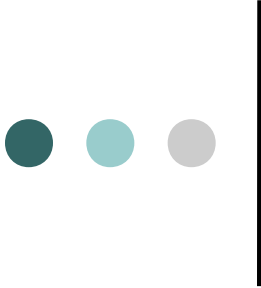
- Pipes sem nome (ou anónimo)
 - Só estão (diretamente *) acessíveis aos processos que os criam e seus descendentes.
 - Novos processos herdam sempre os ficheiros abertos pelo processo pai.
- Pipes com nome ("fifo")
 - São vistos como ficheiros do sistema (têm nome e permissões de acesso)
 - Podem ser usados por qualquer processo através da abertura do ficheiro respetivo.
- (*) É possível partilhar descritores de ficheiros usando o mecanismo de sockets.



Pipes sem nome

○ Programação C:

- `int pipe(int filedes[2]);`
 - `filedes[0]` – descritor de ficheiro para leitura
 - `filedes[1]` – descritor de ficheiro para escrita
 - Usar com funções `read` e `write`, ou aplicar `fdopen`
 - `ssize_t read(int fd, void *buf, size_t count);`
 - `ssize_t write(int fd, const void *buf, size_t count);`
 - `FILE *fdopen(int fd, const char *mode);`
- `man 7 pipe`, `man 2 pipe`



Versão 4 do exemplo de *race conditions*: solução com *pipes*

```
int main() {
    int pid;
    int pfd[2];
    char c;

    pipe(pfd); // uma vez mais, sem verificação de erros...

    pid = fork();
    if(pid==0){
        read(pfd[0], &c, 1); /* wait for parent */
        printf("%d\n", getpid());
    } else {
        printf("%d\n", getpid());
        write(pfd[1], "c", 1);
    }
}
```



FIFO (*named pipe*)

- Um ficheiro *fifo* é um *pipe* com um nome (ou seja, visível no sistema de ficheiros)
 - O mecanismo *fifo* obedece aos mesmos princípios do *pipe* mas pode ser usado entre processos sem qualquer relação entre si.
 - `man 7 fifo`



FIFO (*named pipe*)

- Criação:
 - Na shell: com o comando `mkfifo` ou `mknod`
 - Programação C:
 - `int mkfifo(const char *pathname, mode_t permissions);`
 - `int mknod(const char *pathname, mode_t mode, dev_t dev); // com dev = S_IFIFO`
- Estes comandos só fazem a criação do pipe.
 - Necessário fazer a sua abertura com as funções habituais de abertura de ficheiros.
- `man 7 fifo`



FIFO: exemplo (escritor)

```
#define FIFO_FILE "myfifo"

int main(void) {
    FILE *fp; char readbuf[80];

    mkfifo(FIFO_FILE, 0600); // caso exista, não faz nada

    // aguarda que fifo seja aberto para leitura
    fp = fopen(FIFO_FILE, "w");

    // envia linha de texto
    fprintf(fp, "Hello\n");

    return(0);
}
```



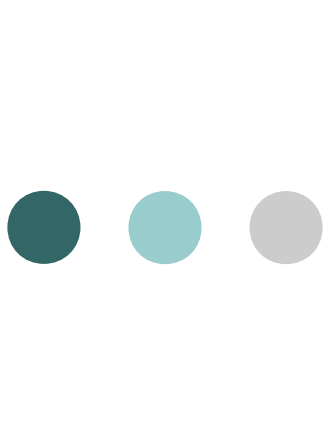

FIFO: exemplo (leitor)

```
int main(void) {  
    FILE *fp; char readbuf[80];  
  
    mkfifo(FIFO_FILE, 0600);  
  
    //aguarda que fifo seja aberto para escrita  
    fp = fopen(FIFO_FILE, "r");  
  
    //aguarda que seja enviada linha de texto  
    fgets(readbuf, 80, fp);  
  
    printf("Received string: %s\n", readbuf);  
    return(0);  
}
```



popen()

- `FILE *popen(const char *command, const char *type);`
- Cria um pipe e um novo processo que por sua vez irá executar, numa *shell*, o comando especificado.
- Retorna um *stream*.
 - Poderá ser de dois tipos:
 - Para leitura do stdout do novo processo
 - Para escrita no stdin do novo processo
 - Esta comunicação é feita através do pipe, i.e., o stream está associado ao pipe criado internamente pelo `popen()`
- Exemplo:
 - `FILE *fp = popen("ls","r");`
- Fechar com o `pclose()` (e não o `fclose()`) - aguarda a terminação do processo criado pelo `popen()`



Comunicação entre processos: filas de mensagens.



Filas de Mensagens (*message queues*)

- Mensagem – bloco de dados com um máximo de `mq_msgsize` bytes
 - `mq_msgsize` é um atributo da fila, definido na sua criação.
- A fila é um *buffer* de mensagens.
 - Mensagens ordenadas por prioridade
 - Mensagens do mesmo nível prioridade tratadas segundo política FIFO
 - Múltiplas tarefas podem enviar mensagens para a fila.
 - Múltiplas tarefas podem receber mensagens para a fila.
 - Cada mensagem só é recebida por uma tarefa



Filas de Mensagens: POSIX API

- `mqd_t mq_open(const char *name, int oflag, mode_t mode, struct mq_attr *attr)`
 - Abre uma ligação a uma fila de mensagens (com criação opcional) e retorna o identificador da fila.

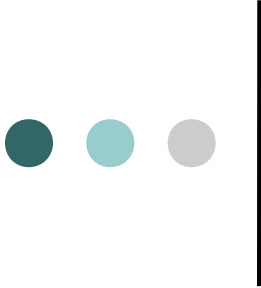
```
struct mq_attr {  
    long mq_flags;      /* Flags: 0 or O_NONBLOCK */  
    long mq_maxmsg;     /* Max. # of messages on queue */  
    long mq_msgsize;    /* Max. message size (bytes) */  
    long mq_curmsgs;}; /* # of messages currently in queue */
```

- `int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned msg_prio)`
- `ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned *msg_prio)`
 - Recebe (e retira) da fila `mqdes` a mensagem mais antiga do grupo de mensagens com o nível de prioridade mais alto.



Filas de Mensagens: POSIX API

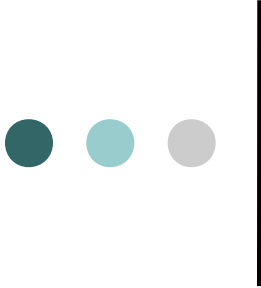
- `int mq_close(mqd_t mqdes)`
 - Fecha a ligação à fila de mensagens `mqdes`.
- `int mq_unlink(const char *name)`
 - Marca a fila para eliminação.
- `int mq_getattr(mqd_t mqdes, const struct mq_attr * attr)`
 - Obtém o atributos da fila `mqdes`.
- `int mq_setattr(mqd_t mqdes, const struct mq_attr * newattr, const struct mq_attr * oldattr)`
 - Permite passar a fila de modo bloqueante para modo não bloqueante e vice-versa. Não é possível alterar os restantes atributos.
- Linux: `man mq_overview`



Filas de Mensagens System V (*legacy*)

- System V API

- `int msgget(key_t key, int msgflg);`
- `int msgctl(int msqid, int cmd, struct msqid_ds *buf);`
- `ssize_t msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);`
- `int msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);`



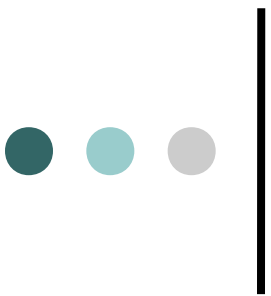
Exemplo 1: envio de mensagem

```
#define MSGQOBJ_NAME    "/myqueue123"
#define MAX_MSG_LEN    70

int main(int argc, char *argv[]) {
    mqd_t msgq_id;
    char msgcontent[MAX_MSG_LEN];

    //variáveis adicionais e analisar argumentos

    if (create_queue) {
        msgq_id = mq_open(MSGQOBJ_NAME,
            O_RDWR | O_CREAT | O_EXCL,
            S_IRWXU | S_IRWXG,
            NULL);
    } else {
        msgq_id = mq_open(MSGQOBJ_NAME, O_RDWR);
    }
}
```

Exemplo 1: envio de mensagem (cont.)

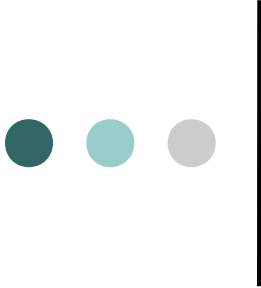
```
/* producing the message */

currtime = time(NULL);
snprintf(msgcontent, MAX_MSG_LEN,
        "Hello from process %u (at %s).",
        getpid(), ctime(&currtime));

mq_send(msgq_id, msgcontent,
        strlen(msgcontent)+1, msgprio);

mq_close(msgq_id);

return 0;
}
```



Exemplo 2: leitura de mensagem

```
#define MAX_MSG_LEN      10000

int main(int argc, char *argv[]) {
    struct mq_attr msgq_attr;
    //... declarações das outras variáveis

    msgq_id = mq_open("/myqueue123", O_RDWR);

    msgsz = mq_receive(msgq_id, msgcontent,
MAX_MSG_LEN, &prio);

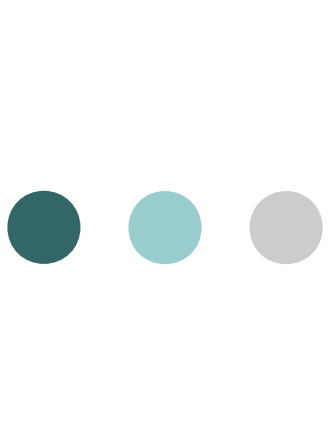
    printf("Received message (%d bytes): %s\n",    msgsz,
msgcontent);

    ...
}
```



Filas de mensagens POSIX: considerações finais

- Permitem implementar, de forma simples, uma política FIFO.
- Envolvem cópia de dados
 - `proc1 => queue => proc2`
- Envolvem chamadas ao *kernel* (*mq_receive*, *mq_send*)
- Mecanismo local (apenas para processos residentes no mesmo computador)



Comunicação entre processos: memória partilhada.



Memória Partilhada (*shared memory*)

- Permite que múltiplos processos possam aceder a um bloco de memória comum.
 - Quando um processo altera o conteúdo dessa memória, todos os processos observam essa modificação.
 - Acesso rápido (igual ao acesso da restante memória do processo), sem intervenção do *kernel* do SO.
 - Principais limitações:
 - Não oferece nenhuma forma de sincronização.
 - Sem sincronização, um processo pode ler os dados enquanto outro processo os está a modificar => potenciais inconsistências.
 - Redimensionamento do tamanho é difícil de implementar.



Memória partilhada (API POSIX)

- Possível criar blocos de memória partilhada com ou sem nome (anónimos).
- Memória partilhada com nome:
 - Criação/abertura de bloco de memória:
`int shm_open(const char *name, int oflag, mode_t mode);`
 - Eliminação:
`int shm_unlink(const char *name);`
 - Exemplo (criação de um bloco de 1024 bytes):

```
int fd = shm_open("/myshm", O_CREAT | O_RDWR, 0600);  
ftruncate(fd, 1024);
```



mmap

○ Permite mapear ficheiros em memória

- `void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);`
 - Algumas *flags*:
 - MAP_SHARED - Updates to the mapping are visible to other processes that map this file, and are carried through to the underlying file.
 - MAP_ANON – (anonymous, sem nome) Usada com a *flag* MAP_SHARED, permite criar um bloco de memória sem ser necessário utilizar o shm_open.
- `int munmap(void *start, size_t length);`
 - The address `addr` must be a multiple of the page size.
 - All pages containing a part of the indicated range are unmapped
 - Closing the file descriptor does not unmap a region.



Utilização de memória partilhada

- Entre processos de diferentes grupos (e.g., diferentes programas):
 - Memória partilhada com nome:
 - `shm_open` + `mmap`
 - `mmap` permite aceder à memória partilhada usando apontadores.

- Entre processos do mesmo grupo:
 - Basta usar memória partilhada sem nome:

`mmap` com `flags = MAP_SHARED | MAP_ANON`

- Exemplo (bloco de 1024 bytes):

```
char *ptr = mmap(NULL, 1024, PROT_READ |  
PROT_WRITE, MAP_SHARED | MAP_ANON, -1, 0);
```




Exemplo: memória partilhada com shm_open + mmap

```
int main(void) {
    char *shmptr; //ou outro tipo qualquer...
    int fd;

    /* Create shared memory object and set its size */
    fd = shm_open("/myshm", O_CREAT | O_RDWR,          S_IRUSR
        | S_IWUSR);
    ftruncate(fd, SHM_SIZE);

    /* Map shared memory object */
    shmptr = mmap(NULL, SHM_SIZE,
        PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

    //Use the memory...
    shmptr[0]=...
}
```



Memória partilhada: System V

- API System V (legado)
 - `int shmget(key_t key, size_t size, int flag);`
 - `int shmctl(int shmid, int cmd, struct shmid_ds *buf);`
 - `void *shmat(int shmid, const void *addr, int flag);`
 - `int shmdt(void *addr);`
 - `(shell) ipcs -m`
- Ainda bastante utilizada, mas evitar em novos projectos: usar norma POSIX.



Exemplo memória partilhada System V (*legacy*)

```
int main(void) {
    int shmid;
    char *shmptr;

    /* Allocate a shared memory segment. */
    shmid = shmget(IPC_PRIVATE, SHM_SIZE,
        IPC_CREAT | 0666);
    /* attach the shared memory segment. */
    shmptr = shmat(shmid, 0, 0);

    //use the memory...

    /* Detach the shared memory segment. */
    shmdt (shared_memory);
    /* Deallocate the shared memory segment. */
    shmctl(shmid, IPC_RMID, 0);
}
```