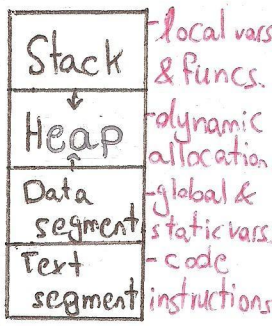
Processes [1]

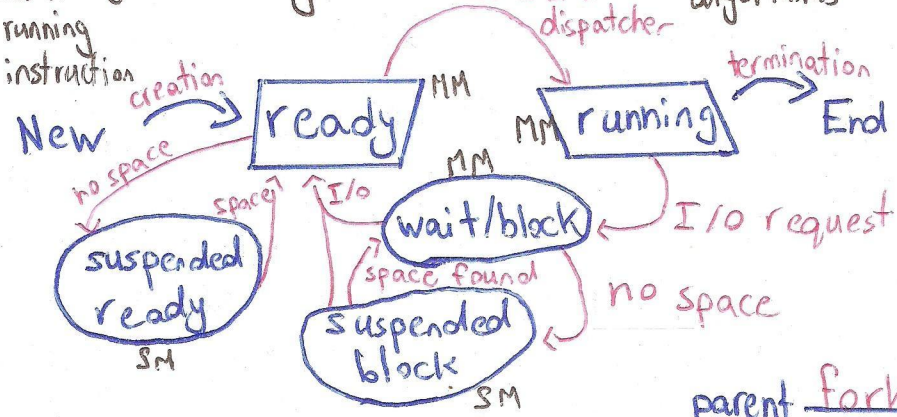
- **Process:** program in execution. prog.c —compiled→ prog.out —run(./...)→ process:



- **Implementation:** Table of processes (array of structs) one entry per process.

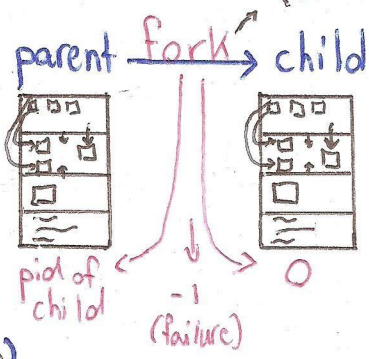
pid — process ID
ppid — parent's ID
PC/IP — program counter, pointer to currently running instruction
stack pointer — pointer to address of stack in memory
text & data segments
priority — based on scheduling algorithms
files opened
. . . .

... status:



MM: main memory
SM: secondary memory

- **Creation:** UNIX Boots ⇒ init (of pid = 1) runs
  any others must be created by duplication

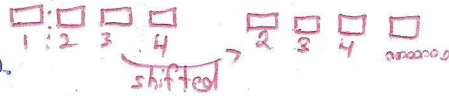


- **Identification:** •getpid(); caller , •getppid(); parent
- **Termination:** -normal { -returned (finished execution) -exit(); }
  -abnormal { -killed by another process (one with priviledges).

-void exit(int status); status: 0: normal (equivalent to return();)
else: message from child up to parent.

-int wait(int * status); returns: -1: no children alive.
: pid of exited child.

NOTE:
exit(a); → wait(&a);
int = 4 bytes



-<sys/wait.h>:

int WIFEXITED(int status) return nonzero for normal termination.
int WEXITSTATUS(int status) return the low order 8 bits of child's exit status.
int WIFSIGNALED(int status) return nonzero if child terminated for an unhandled signal.
int WTERMSIG(int status) return signal number of process that terminated it.

int waitpid(int pid, int * status, int options)
- pid: -1: no specific child; WAITANY: same
- options: null: ≈wait; WNOHANG: parent isn't blocked; WUNTRACED: request status info. about stopped & terminated children.

- Zombie process: Died before parent calls wait().
- Orphan process: Process of a dead parent (becomes child of init)

• Execution: <unistd.h> has the family of exec functions:
once reached, a process forgets its current tasks and executes code from the given file (without returning).

- int execl (char * filename , char * arg0 , char *arg1, ...);
  - int: -1: failure, file not found
  - filename: file to execute
  - arg0: filename without path
  - arg1, ...: arguments
  - ex: execl("/bin/ls", "ls", "-l", "-A", NULL);
- int execv (char * filename, char * argv[]); same but in vector form
- Adding a "p" to the name (execlp, execvp) will cause the prog to check inside "PATH" which is where newly added programs place a path to them upon instalation => less chance of file not found.

Note: void main(int argsCount, char *argv[])
for these we need atoi (ASCII to integer)

Command Line arguments:

• void main (int argc, char *argv[]);
- argc: count
- argv[]: vector (array)

Conversion: atoi (char * n) returns integer

forks & B trees:

A && B: A false => B never evaluated.
A || B: A true => B never evaluated.

Inter-process-Communication (IPC)

- same machine (this course)
- diff. machine (next course)

- pipes - message passing - signals - shared memory slots

• FIFO Buffer in the main memory of fixed size (4kB or 8 kB)
• one-sided: trying to read/write from one process => loss of data.
• reading from empty pipe => blocked till someone writes (get data) / all writers closed
• writing to full pipe => blocked till someone reads

• Creation:
int fd[2];
int pipe(fd); -1 fail, 0 success
write in fd[1], read in fd[0] => usually we define W 1 and R 0.
fork(); -> • all vars duplicated except for the pipes.
• pipes must be created before forking.

• A process not using a side of the pipe must close it with:
int close(f[W]); int close(f[R]);
• Writing: write (fd[W], buffer, size); close(fd[W]);
char *src = "..." ~ strlen(src);
• Reading: read (fd[R], buffer, size); close(fd[R]);
char dest = char[100];

File Descriptor Redirection:

Process created => it gets a "table of descriptors"

0 <- stdin (keyboard)
1 <- stdout (screen)
2 <- stderr
5
:

To change them, we use an int fd[2];
where fd[0] read
fd[1] write
(make it easier with #define W 1
#define R 0)

dup2(fd[0], 0); standard input now fd[0].

dup2(fd[1], 1); standard output now fd[1].

Note: close everything everywhere.

Signals (software interrupts)

- Kinds of signals:
  - program errors
    - invalid memory address
    - dividing by zero.
  - interrupt (ctrl-\)
  - terminate (ctrl-c)
  - I/O that can't be done
    - writing to full pipe
  - child termination
  - timer expiration
  - process calls to "kill" or "raise" self.

- Categories:

errors - external events (I/O) - explicit requests ("kill") - Miscelanious Signals (process communication)

- Each has an ID number (int)
- There are 31 Signals in <Signal.h>

- Signal Receiving: 3 choices:
  1. Ignore it (if possible)
  2. Default Action
  3. Handle it (if possible)

- Termination Signals
  - SIGTERM: (ctrl-\) can be handled
  - SIGKILL: can't be handled
  - SIGINT: (ctrl-c) can be handled

- Alarm Signal
  - SIGALRM - can be handled
    - default: terminate

- Error Signals
  - SIGFPE: floating point exception
  - SIGILL: Illegal Instruction (trying to execute a privileged instruction)
  - SIGSEV: segmentation violation (trying to read/write outside allocated memory)

- I/O Signals
  - SIGIO
  - SIGURG
  - default action: ignore

  Both can be handled.

- Job Control Signals
  - SIGCHLD: sent to parent if child dies
    - can be handled
  - SIGSTOP: stops a process, can't be ignored.

Other Purpose Signals:
- SIGUSR1
- SIGUSR2
  - communication
  - must be handled.

• Signal Handling:

<signal.h> has `Signal(int signNB, sighandler_t ACTION);`

signal to handle

- SIG_DFL default
- SIG_IGN ignore
- name of a user defined function (myHandler)

`void myHandler(int sigNB);`

• Sending A Signal:

- success - `int kill(int pid, int SIGINT);`
- fail

• Waiting For A Signal:

<unistd.h> has `int pause();`
that suspends execution till a signal arrives.

• Alarm:

`int alarm(int nsec);` lets a process interrupt itself in the future.

- time expires => SIGALRM (can't be ignored)
- called twice => last call is taken into account
- set off with `alarm(0);`

Named Pipes (FIFO): have a name and permanent existance.

- can be used by independant processes
- capacity ~40KB
- has R and W sides
- Opening W side is blocked till someone opens the R side (same for R).

• Creation:

Shell

```
mkfifo name -m 0666
```
(perms.)

Program <fcntl.h>

```
int mkfifo(char *filename, mode_t mode);
```
(mode: perms, e.g. 0666)

0 success
-1 fail.

• Using:

```
int fd = open("filename", flags);
```
(char *) (int)

↳ O_RONLY read only
↳ O_WONLY write only

Now we can use:
- read(fd, &x, sizeof(int));
- write(fd, &x, sizeof(int));
- close(fd);

Shared Memory Segment:

#include <sys/types.h>
<sys/ipc.h>
<sys/shm.h>

step1: getting the Key

Method 1: do it yourself: key_t k = 1234...
Method 2: Generate one: key_t k = ftok(char *path, int ID);
(usually "." and 'a')
Method 3: Use IPC_PRIVATE so the system generates a private one.

step2: request memory (here)

```
int shm_id = shmget(k, sizeof(ObjectToShare), );
```
↳ 0666 (if only using)
↳ IPC_CREAT | 0666 (if creating and using)

step3: attach a pointer to the memory

```
Object p = (Object *) shmat(shm_id, NULL, 0);
```

Now we can use p as a pointer.

(can be int, struct, etc...)

step4: detach pointer when done

```
Shmdt((void *)p);
```

step5: de-allocate & remove memory:

```
Shmctl(shm_id, IPC_RMID, NULL);
```

Note: Do steps 3 & 4 on all sides.
Do step 5 only once.