- Processus : Entité dynamique représentant l'exécution d'un programme sur un processeur
- Point de vue OS : Espace d'adressage (mémoire, données, code) + état interne (compteur exec, fichiers ouverts, . . .)
- Programme = Code + données, Processus = Programme + mémoire
- Pseudo parrallélisme = Chacun son tour (fonctionne grace aux interruptions matérielles)
- pause() = attente d'un signal
- Attente de la fin de l'un de des fils : pid_t wait(int *ptr_etat) (retour = pid fils qui a terminé + code de fin stocké dans ptr_etat)
- Attendre la fin du fils : pid_t waitpid(pid_t pid, int *ptr_etat, int options)
- Processus zombie : terminé mais le père n'a pas appelé wait()
- code=execl("/bin/ls", "ls", "-a", 0);
- Signaux :
  - SIGINT
  - SIGUSR
  - SIGALARM : fonctionne avec alarm(int n) qui retourne le temps restant ou 0
  - SIGCHILD = signal auto lors de la fin ou suspension d'un fil à son père (si sigaction défini)

Threads :

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void *print_message_function( void *ptr );
main(int argc, char* argv){
     pthread_t thread1, thread2;
     const char *message1 = "Thread 1";
     const char *message2 = "Thread 2";
     int  iret1, iret2;
    /* Create independent threads each of which will execute function */
     iret1 = pthread_create( &thread1, NULL, print_message_function, (void*) message1);
     if(iret1){
         fprintf(stderr,"Error - pthread_create() return code: %d\n",iret1);
         exit(EXIT_FAILURE);}
     iret2 = pthread_create( &thread2, NULL, print_message_function, (void*) message2);
     if(iret2){
         fprintf(stderr,"Error - pthread_create() return code: %d\n",iret2);
         exit(EXIT_FAILURE);}
     printf("pthread_create() for thread 1 returns: %d\n",iret1);
     printf("pthread_create() for thread 2 returns: %d\n",iret2);
     /* Wait till threads are complete before main continues. Unless we  */
     /* wait we run the risk of executing an exit which will terminate   */
     /* the process and all threads before the threads have completed.   */
     pthread_join( thread1, NULL);
     pthread_join( thread2, NULL);
     exit(EXIT_SUCCESS);}

void *print_message_function( void *ptr ){
     char *message;
     message = (char *) ptr;
     printf("%s \n", message);}
```

Signaux :

```c
#include <signal.h>
void handler(int sig) {
     /* nouveau gestionnaire */
     printf("signal SIGINT recu !\n");
     exit(0);}
int main() {
    struct sigaction nvt,old;
    memset(&nvt, 0, sizeof(nvt));
    nvt.sa_handler = &handler;
    sigaction(SIGINT, &nvt, &old);
    /*installe le gestionnaire*/
    pause ();
    /* attend un signal */
    printf("Ceci n'est jamais affiche.\n");;
}
```

Traitement erreur :

```c
    waitpid(pid, &e, 0);
      if(WIFEXITED(e) && WEXITSTATUS(e) == 0)
        exit(1);
```

Pipe déterminer taille buffer :

```c
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

int main(int argc, char *argv[]){
  int fd[2];
  pipe(fd);
  int BUFSIZE=atoi(argv[1]);
  char input[BUFSIZE];
  int i;
  for(i=0; i<BUFSIZE; i++){
    strcat(input, "1");
  }
  write(fd[1], input, strlen(input)+1);

  //La taille max est 65536
  return 0;
}
```

1 Vérrouillage

- F_ULOCK : dévérr

- F_LOCK : verr exclusif
- F_TLOCK : verr exclusif avec test (ne bloque jamais mais retourne une erreur)
- T_TEST : test seulement

taille = 0 pour le fichier complet, > 0 nb octets après la position, et inversement

```
#include <unistd.h>
int main (void) {
  int fd;
  fd = open("toto", O_RDWR); /* doit exister */
  while (1) {
    if (lockf(fd, F_TLOCK, 0) == 0) {
      printf("%d: verrouille le fichier",
         getpid());
      sleep(5);
      if (lockf(fd, F_ULOCK, 0) == 0)
    printf("%d: deverrouille le fichier",
           getpid());
      return;
    } else {
      printf("%d: deja verrouille...",
         getpid());
      sleep (2);}  } }
```

Informations specifiques à un thread : Pile Registres Priorité (d'ordonnancement) Données spécifiques Liste des signaux bloqués

2 Thread

- pthread t : équivalent du pid t (c'est une structure opaque)
- pthread t pthread self () : identité du thread courant
- int pthread equal (pthread t, pthread t) : test d' égalité
- int pthread create(identité, attributs, fonction, argument) ;
- Terminer le thread courant : void pthread exit(void *retval) ;
- Attendre la fin : int pthread join (pthread t, void **)
- int pthread detach (pthread t)
- On passe deux argument à la fonction en définissant une structure

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 8

typedef struct data {
    int thread_id,sum;
    char *msg;
} data_t;

data_t data_array[NUM_THREADS];
```

```c
void *PrintHello(void *arg) {
     data_t *mine = (data_t *)arg;
     sleep(1);
     printf("Thread %d: %s Sum=%d\n",
     mine->thread_id,
     mine->msg,
     mine->sum);
     pthread_exit(NULL);
}
```

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 3
void *travail(void *null) {
     int i;
     double result=0.0;
     for (i=0; i<1000000; i++)
     result = result + (double)random();
     printf("Resultat = %e\n",result);
     pthread_exit(NULL);}
int main(int argc, char *argv[ ]) {
    pthread_t thread[NUM_THREADS];
    pthread_attr_t attr;
    int rc, t;
    /* Initialise et modifie l'attribut */
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr,
    PTHREAD_CREATE_JOINABLE);
    }
```