Fuctions:

getpid() (<unistd.h>) getppid()

fork() (<unistd.h>)

exec*() (<unistd.h>)

```
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ..., char * const envp[]);
int execv(const char *path, char * const argv[]);
int execvp(const char *file, char *const argv[]);
```

path: 应动程序的名称目标路径; arg: 应动程序所需参数, 以NULL结束

return: 0 S, -1 F

前l: 可变参数 NULL结束

前p: path 需完整路径, 从环境变量中寻找命令.

前l: arg is [] 结束, (x NULL结尾.

前e: 将环境变量传给替换的进程

wait() (<sys/types.h>) wait one of the children

waitpid() (<sys/wait.h>) wait for a particular child.

ps aux | grep defunct zombies process.

Banker's Algorithm

[Avail] = [Free Resources]

Add all nodes to UNFINISHED

do { done = true Foreach node in UN { if [~~Request~~ $_{node}$] <= [Avail]) remove

[MAX$_{node}$] - [Alloc$_{node}$]

Deadlock detect Algorithm.

[Avail] = [Free Resources]

Add all nodes to UNFINISHED

do { done = true; Foreach node in UNFINISHED { if [Request$_{node}$] <= [Avail]) { removeen node from UNFINISHED [Avail] = [Avail] + [Alloc$_{node}$] done = false } } untill done)

Detect deadlock: terminate thread, preempt resources, Roll back actions, ignore

Prevent: Infinite resources, No sharing of resources, don't allow waiting, request at first

Four requirements for Deadlock:

Mutual Exclusion, Hold and wait, No preemption, Circular wait.

IPC Sychronization:

Guarantee mutual exclusion;

Uphold the correct sychronization among processes, dead lock-free

Dining philosopher

```
void take_chop(int i)
    wait(&chop[i]);
void put_chop(int i)
    post(&chop[i]);
void philosopher(int i){
    while(TRUE){
        think();
entry -> take_chop(i);
        take_chop(i+1)%N);
S ->    eat()
exit -> put(i)    put((i+1)%N) } }
```

```
void captain (int i) {
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) { state[i] = EATING
        post(&p[i]) }
```

Final Solution:

Shared object

```
#define N 5
#define left (i+N-1)%N)
#define RIGHT(i+1)%N)
int state[N]
semaphore mutex=1
          P[N]=0;
```

Main

```
void philosopher(int i)
    think();
    take_chops(i);
    eat();
    put_chops(i);
}
```

Section Entry

```
void take_chops(int i)
    wait(&mutex)
    state[i] = HUNGRY
    captain(i)
    post(&mutex)  wait(&p[i])
```

Exit

```
void put_chops(int i)
    wait(&mutex)
    state[i] = THINKING
    captain(LEFT)
    captain(RIGHT)
    post(&mutex
```

```
void sem_wait(semaphore *s){
    disable_interrupt();
    *s = *s - 1;
    if (*s < 0){
        enable_interrupt();
        sleep();
        disable_interrupt();
    }
    enable_interrupt();
}
```

wake up()

post

```
Void producer(void){
    int item;
    while(TRUE){
        item=produce_item()
        wait(&avail)
        wait(&mutex)
        insert_item(item)
        post(&mutex)
        post(&fill)
    }
}
```

```
void consumer(void)
    int item;
    while(TRUE){
        wait(&fill)
        wait(&mutex)
        item=remove_item()
        post(&mutex)
        post(&avail)
    }
}
```

#define N 100 semaphore mutex=1 avail=N fill=0

System call

Process control: fork(), ~~get~~exit(), wait()

File manipulation: open(), read(), write(), close()

Device manipulation: ~~int~~ioctl(), read(), write()

Information Maintenance: getpid(), alarm(), sleep()

Communication: pipe(), shmget(), mmap()

Protection: chmod(), umask(), chown()

Inter-process Communication (IPC) → one-way, 2 processes, some ancestor

SIGNAL, Shared Object (Pipe, FIFO (Named pipe), Semaphore

In kernel level, User-space not sharable

Race condition: shared object + multiple processes + concurrently.

Mutual Exclusion, Critical Section

Entry & Exit implementation - Requirements:

Mutual Exclusion: No two process goes at a time.

Bounded waiting: Finite waiting time

Progress: At least CS is used by someone.

Achieving Mutual Exclusion

Lock-Based: Spin-based lock, Sleep-based lock

~~Sleep-Based~~ Lock-free.

#0 - disabling interrupt for the whole CS

Aim: disable context switch

Uni-core. Correct but permissible

Multi-core: Incorrect.

#1 Basic Spin Lock (busy waiting)

```
while(TRUE){
    while(turn != 0){
        /* busy waiting */
    }
    critical_section();
    turn = 1;
    remainder_section();
}
```

```
while(TRUE){
    while(turn != 1){
        /* b w */
    }
    critical_section();
    turn = 0;
    remainder_section();
}
```

Correct but wastes CPU resources.

#2 Spin Smarter suffer from Priority Inversion

```
int turn;
int interest[2] = {FALSE, FALSE};
void lock(int process){
    int other;
    other = 1 - process;
    interested[process] = TRUE;
    turn = other;
    while(turn == other && interested[other] == TRUE);
}
void unlock(int process){
    interested[process] = FALSE;
}
```

```
level[N] = {-1};
waiting[N-1] = {-1};
for(int i=0; i<N-1; i++){
    level[i] = i;
    waiting[i] = i;
    while(waiting[i] == i && there exists k≠i level[k] ≥ i))
}
//CS
level[i] = -1 //exit
```

N

#3 Sleep-Based Lock: semaphore

Producer-consumer