Reader-Writer.

Shared Object.
Semaphore mutex = 1.
semaphore write = 1
int readcount = 0

Writer.
do {
wait(& write)
// WRITE
post(& write)
} while(true).

Reader
do {
wait(& mutex);
readcnt ++;
if(readcnt == 1) wait(& write);
post(& mutex)
// READ.
wait(& mutex);
readcnt --;
if(readcnt == 0) post(& write);
post(& mutex);
} while(true).

Race condition:
shared Object.
Multiple Process/Thread.
Access concurrently.

Critical Section Check.
Mutual Exclusion.
Bounded Waiting (Starvation)
Progress (Deadlock)

After Deadlock Attribute
Allow Recover.
Prevent
Ignore.

After Deadlock.
Terminate Process.
Preempt w/o killing.
Rollback action

Deadlock.
Mutual Exclusion.
Hold And Wait
No preemption.
Circular Wait

Prevent Deadlock
Infinite Resource.
No sharing
No waiting: allow Preempt
Request All need at beging.
Force Request order.

Achieve MR
Lock. — Spin-based
Lock-Free. — Sleep-based.
CAS...
Versioning.

#0: Disable Interrupt for whole CS.
unicore: correct but not permissible. (user x kernel √)
multicore: incorrect. (process on other core). may still change shared object.

#1. Basic Lock.
while(True){
while(turn != 0);
CS
turn = 1
reminder()
}
correct.
OK for short spin < context switch.
strict alternation.

#2 Spin Shaster.
correct.
priority inversion problem.

#3. Semaphore.
need interact with scheduler
uninterruptable section entry/exit.
used as ~~operator~~ synchronization ~~primitive~~ primitive.

Banker Algo
[Avail] = [FreeResources]
All node add to unfinished.
do {
done = true.
foreach node in unfinished {
if. [MAX] - [Alloc] ≤ [Avail]) {
remove node from UNFINISHED.
[Avail] = [Avail] + [Alloc]
done = false.
}
}
} until(done)

[request] node.
node node vector

IO Ctrl: IO instruction. mem-mapped IO.
IO trans: Programed IO. DMA.
HDD: sector. track. cylinder.
Disk Time: Queue. ctrl. seek Time. rotational Delay. Transfer.

Fork.
1. copy PCB
2. kernelspace update.
parent - Add new child.
child - update PID. running time. ptr to parent.
3. user space update.
copy memory context (COW).
4. finish.
update parent/child return value.

Don't change (4)
PC/Register Code, Memory, Opend File
Change (3)
return value.
PID. parent.
running time.
time locks.

File size
$12 \cdot 2^x + 1 \cdot \frac{2^x}{4} \times 2^x + 1\left(\frac{2^x}{\varphi}\right)^2 \cdot 2^x + 1 \cdot \left(\frac{2^x}{\varphi}\right)^3 \cdot 2^x$
$= 12 \cdot 2^x + 2^{2x-2} + 2^{3x-4} + 2^{4x-6}$
2^x: block size.

4 Fundmental. OS.
Thread
Address Space.
Process.
Dual mode / Protection

IPC
Shared File.
Pipe / FIFO Named Pipe.
Shared File
shared Memory
Semaphore

Message Passing
Signal.
Message Queue.
Socket
MPI Library

Real ☐ User+Sys
<: Multi. process. Multi. core.
>: IO-intensive.
Scheduling.
Mode. Change Overhead.

binary semaphore ☐ mutex
semantic: resource protection / resource assignment.
unlock: By container / By Anyone.
Terminate: will unlock. / will not add up.

Parallel/concurrency
Concurrency: structure Coherence.
Parallel: Running status. Hardware. multiple units.

Thread/Process
Process: Program in execution. Port, share memory. More context switch. Less. Execution environment with restricted right. instance of a program. Addrspace √ have its × don't.
Thread: Part of process. share memory. Single unique execution context. state.

LifeCycle
Init → Ready → Running → Finished
Waiting.
Just Forked → Ready → Running → Terminated / Zombie
Init → Ready → Running → Blocked.

Save of CS
Thread: TCB. reg. PC. stack ptr.
Process: + +PCB. file desc table.

User Thread
Same ~~task~~ task. different data.
different independent task (CPU/IO).

My sys rule
1. Don't trust user stack. efficiency.
2. Reg. scheduling can easy switch. no need to save kernel stack every time.

Dining Philosopher

shared Object

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

Section Entry

```
void take_chopsticks (int i) {
    wait (&mutex);
    state[i] = HUNGRY;
    captain[i];
    post (&mutex);
    wait (&p[i]);
}
```

Main Function

```
void philosopher(int i){
    think()
    take_chopsticks(i)
    eat()
    put_chopsticks(i)
}
```

Section Exit

```
void put_chopsticks(int i){
    wait(&mutex);
    state[i] = THINKING;
    captain(LEFT)
    captain(RIGHT)
    post( &mutex)
}.
```

Semaphore.

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

void post( semaphore *s) {
    disable_intr();
    *s=*s+1;
    if (*s <= 0){
        wakeup();
    }
    enable_intr();
}
```

Helper.

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

```
typedef struct{
    int value;
    list process_id;
} semphore;
```

Producer - Consumer.

Shared Object.

```
#define N 100
semaphore mutex=1;
semaphore avail=N;
semaphore full=0;

// full: occupied slot
// avail: empty slot
```

Producer.

```
void producer ~~(void)~~ {
    int item;
    while(TRUE){
        item= produce();
  ⇒     wait( &avail);
  ⇐     wait ( &mutex);
        insert_item (item);
        post ( &mutex);
        post ( ~~&avail~~ &full);
    }
}
```

Consumer

```
void consumer{
    int item;
    while(TRUE){
        wait (&full);
        wait (&mutex);
        item = ~~get~~ remove_item();
        post( &mutex);
        post ( &avail);
        // consume
    }
}
```

① using only avail
如果先consumer 就Get null了.
② 改顺序.
Buffer满了. producer 等avail进入 sleep. consumer等mutex. 就 deadlock了.

Spin smarter. (Peterson).

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

turn = 互斥 Mutual Exclusion.
interest = 回避 strict alternation.
问题: priority inversion problem

← ~~谁~~ 最后一个进入CS.

P0 P1
lock
turn=0
...
CS

lock.
interest[1]=TRUE
turn=1.
~~while~~ while{
... //waiting
~~wait~~

CS
unlock.
interest[0]=FALSE

//wait done
CS

P0 P1. P1
lock
interest[0]=TRUE
turn=0.

turn=1
while →
no wait
CS

lock.
interest[1]=TRUE
turn=1.

while ...
wait.
⋮

Riter Algo.

```
level[N] = {-1}
last_to_enter[N-1] = {-1}
```
↑ waiting.

~~waiting~~: 等候房间

lock:
i ← ~~pid~~ process no.
for ℓ from 0 to N-1 exclusive.
level [i] ← ℓ.
last_to_enter [ℓ] ← i
while last_to_enter[ℓ] = i and exist k≠i, level[k] ≥ ℓ: 阻塞
wait

1. 前方没有别的线程在更高房间内等待
2. 别的后来者进入我的房间. 改了waiting. 被通过了

unlock
level [i] = -1. 当进入CS. 就回退出所有房间.