```
int fd[3];
int count;
if(pipe(fd) != 0)
    perror("...");
count = write(fd[2], "Hello", 6);
printf("%d", count);
return(0);
}
```

so write on pipe fails, so returns -1.

10th Oct 19

⟹ Synchronisation — CSP — Semaphores.

└ critical section problem.

Semaphore is implementation support in Linux for synchronisat^{n} issues.

→ In multithreading, we are more interested in parallelizat^{n}, but there is no guarantee on consistency of data.

data manipulat^{n} ≠ safe. i.e. consistent.

→ Whenever data is shared across threads, more than one thread is manipulating the same data, there would be a conflict.

→ A wants to access a, b, c

B " " " c, d, e

here 'c' is common.

when A & B want to read, then no conflict, otherwise there would be a conflict.

→ Commit saves the changes made permanently.

→ When a shared data is being access, let it be accessed

by using a 2 phase lock.

→ consider 'x'

LOCK(x)
UNLOCK(x)

≡ semaphores.

Thread 1 has gained exclusive lock over 'x', so now none of the other threads can access X. (i.e either read lock or shared lock over X)

→ But in RL, there are many items that are being shared.

→ On the algo front, it is referred to as CSP.

→ Dining philosophers problem.

forks[5] = 1/0 — resource not available

resource available

assume round robin access. When a ~~th~~ philosopher drops the resource, it would be like unlocking.

→ Portion of process defn. which has shared data item being accessed or there is a need for data consistency is called CSP, one of the algo soln is Peterson's algo.

→ Threading and synchronisation go hand in hand.

→ process could have the following blocks —

entry section — checks the conditions to be followed for a process to enter into its critical section. ≡ locking logic.

critical "

exit " — equivalent of unlock.

remainder " — non conflicting data access

✓ wherever sharing is required.

✓ i.e. non critical region of code part where sharing is not required.

→ If a process is in critical section, no other process will be allowed to be in their critical section. This is called mutual exclusion. There would be exclusive access. When one process is in CS, ~~the rest are~~ i.e. not until the end of main but till those instructions which have X involved, otherwise it would become like a non-preemptive kernel.

→ This is at the atomic level i.e. at the level of data items.

↳ This access should be for a limited time only as the other processes are in wait queue to enter into their critical section. So it should ensure in fair-share access.

→ Bound (wait).

→ Producer-consumer simulation.

Assume there is a variable BS 10, buffer which is an array which contains items being produced & consumed i.e. a circular array, index called 'in' and 'out', counter

for producer process (i.e. write position).

producer - produce an item to be ~~pro~~ consumed.

This would be an infinitely running process.

Producer

```
while (True)
    while (ctr == BS)
        ; // busy wait
          // equivalent to entry section.
    buf[in] = item;
    in = (in+1) % BS;
    counter++;
```

Consumer

```
while (True)
    while (ctr == 0)
        ;
    item1 = buf[out];
    out = (out+1) % BS;
    counter--;
```

→ Let ctr=5

T_1 (prod.)	T_2 (cons.)
(1) $r_1 = ctr$; $r_1 = r_1 + 1$; } // counter++; (3) $ctr = r_1$;	$r_2 = ctr$; (2) $r_2 = r_2 - 1$; $ctr = r_2$; (4)

(this is execution in ||. if we run prod. then consumer, then it is serial mode

(1) $r_1 = 5$ $r_1 = 6$	$r_2 = 5$ $r_2 = 4$ (2)
(3) $ctr = 6$	$ctr = 4$ (4)

so racing is happening here. As the order of execution changes, the value of the shared variable counter would vary.

14/Oct/19

⇒ CSP solution — Peterson's algo.

* race condition.
* when we have ++ and — on the same variable at the same time, then it would yield a wrong result.
* interleaved execution.
* Serial execution is, then throughput, η, etc will all be affected
* How to simulate ||ism in a serial environment, that is what the RR, etc do.
* ~~Is it fair that~~
* if there are n processes, then $n!$ serial schedules are possible.
* But in parallel schedule
* Serial sched. are safe because of data consistency.
* focus of serializability of schedules.
  || schedule = serial schedule.

i.e. whenever there is a clash, then run in serial fashion, otherwise run in a ll lel fashion.

* Access to shared variable must happen in a synchronized manner.

* Socializability - a parallel sched. whose sched will show the same effect as serial schedule.

* Semaphores are preferred.

* Prob. with bin locking mechanism

* Result equivalence is less preferred than conflict equi.

problem - how many states can we keep checking, it is exhaustive. (this is a more empirical way).

if there are conflicting operations (i) inst should come from 2 diff. transactns, ii) operating on atleast one data item in common which they are manipulating. iii) should be (w,w) or (w,r) or (r,w)

Two sched. are conflict equi., if the order in which ordering of operations is same.

$r(x)$, $w(x)$ $\quad$ $r(x)r(y)$ $r(z)$ $w(x)$

these are conflict equivalent.

* These are necessary conditions to specify CSP solution (all should be followed)
  - MUTEX (mutual exclusion)
  - progress
  - B-wait (bounded wait)

These three can formally prove mathematically that

* MUTEX - I can't have two processes active in the CS at the same pt of time. i.e. access to Critical Section is mutually exclusive.

* progress - if no process in CS, & there are a few processes who want to go to CS, which process should be allowed, i.e we look at the processes which are not in the remainder section. as the others have never gained the CS. This would be fair-share scheduling. CSP should be fair to all processes (assuming priority is same)
* Bounded wait - there should be a bounded to the waiting time in the CS.

15/10/19

→ Critical section problem:

4 blocks: entry section
critical section
exit section.
remainder section.

1) Mutex: "in their cs"
2) Progress:
3) Bounded wait

→ Peterson's algo for CSP [2 P_i setup]

The desire to enter CS is expressed by flag[i/2] to true.
These two var. (i.e. turn, flag) decide the entry to entry section (?)

Turn — indicates whose turn it is to be in CS. So we should set turn (it is an atomic var) to enter CS.

1 or 2

Here, assume flag[i] = true, and turn = 1, then i can get into CS. Even if flag[j] = true, j can't get into CS as it is i's turn in CS.

* Busy wait - resource is busy, so we have to wait.

P_i

```
do
{
    flag[i] = TRUE;
    turn = j; // goes to busy wait
}
while (f[j] && turn = j); // simulates keep waiting for lock.
    "CS"
(remainder section)
flag[i] = False;
} while (true);
```

P_j

```
do
{
    flag[j] = true;
    turn = i;
    while (f[i] && turn = i);
        "CS"
    flag[j] = false;
} while (true);
```

→ Next step is to prove that mutex, progress & BW is satisfied.

→ Proof that mutex is satisfied.

proof by contradiction.

~(Mutex);

i.e. flag[i] = f[j] = True (allowed). (desire to enter CS)

turn = i/j

turn cant be set as i & j at the same time as it is an atomic instruction.

"turn" is simulating the lock behaviour.

→ Proof for progress.

Inference of progress & BW go together.

i.e. proof processes are progressing after waiting for bounded time.

Assume $P_i \neq CS$; // P_i is prevented from progressing.

f[j] = true ∧ turn = j // prevents P_i from going to CS.

(1) ~~f[i] =~~

i) Assuming f[i]=true

1) if f[j]=false, then don't look at turn, P_j can't move to CS, then P_i progresses.

ii) Assuming f[j]=true.

then turn can be either i or j.

1) If flag[j]=T and turn=j, then P_j progresses.

After P_j is done with CS, then flag[j]=F, if it further requires CS, then it would again be flag[j]=T. and then turn=i, so P_i progresses.

The bound here is after P_j is done with CS, and then setting flag[j]=F, depicting that P_i only needs to wait for one clock cycle/time instant

2) If flag[j]=T and turn=i

→ 'n' process setup.

max bound = (n-1) clock cycles.

i.e. (n-1) wait

* sem_wait — lock

sem_post — unlock

like posting saying their CS is over.

* pthread_mutex_lock — fails to look at multiple resource scenario

pthread_mutex_unlock } this is precisely a boolean setup. (0/1)

* we can issue sem_wait until ~~there are~~ resources are present.

* sem_wait

```
wait(s)          // where s is a resource.
{
   while (s<=0); // it we have has to wait
   s--;          // release will happen when someone does
}                // locking, i.e no one else will s++;
                 // get it now.
```

* sem_post

```
s++;
```

* the order of calling and what is being called would make a difference.

21/10/19

⇒ Semaphores.

Peterson's n P_i setup.

after 'n' cycles, ~~re~~ reentry happens.

* test-set : test status of lock or flag variable and set it.

```
bool test_set(bool *target)
{ ~~bool~~ return value      // lock variable.
   bool rv = *target; // current status is remembered.
   *target = TRUE; // setting the lock, unconditional
   return(rv); // you are returning the old status.
}
```

lock = 0;
do {
    while (ts(&lock)); // busy wait
                       // bounded wait

    "CS"

    lock = false;
} w(T);

* cs (lock, exp, new_value) → 0 for lock to happen

eg. when process 'i' is in CS.

do {
    waiting[i] = True; // process waiting to go to cs.
                       // =1, then desire to go to cs., similar to w[i]
    key = True;
    while (waiting[i] && key)
        key = test_set (&lock); // if lock is initially free and key, w[i] = T, then key is updated, lock gets set to 1, which simulates entry to CS.
    // i' moves to "CS"

    waiting[i] = False;
    j = (i+1) % N;
    while (j != i && !waiting[j])
        j = (j+1) % N;   // if 1 is not there, then we check if 2 is there and so on.
    if (j == i) // if bound wait has elapsed, i' re-enters
        lock = False;
    else
        waiting[j] = False;
} while (T);

* Implement Dijkstra's, Decker's and Peterson for CSP.
Pick one more problem from the textbook.
Producer, consumer // simulate this
* Stallings, Schilbertz

22/10/19.

⇒ Classical problem of synchronization.

<semaphore.h> semaphore based implementation

sem_t

* race condition.
* sem_t
→ readers & writers
→ dining philosophers
→ producer consumer
* simulation of lock, unlock

sem_wait, sem_post *, sem_init
|| wait || post initialization of semaphore.

(s<=0);

wait(s)
{ while(s<=0);
→ //CS here
s--; //more of a resource.
}

s++ in post def:≡ unlock

```
int data=0, rcount=0;   // all read - no problem, but everytime thread reads, increase rcount
                        // even if there is a single write, block read and then read.
                        // when they are reading and there is a write, wait for read to complete and then write.
sem_t mutex;
    will help in proper sync of processes i.e. r & w.
sem_t writeblock;       // there are 'm' writer threads and 'n' reader threads
    // binary resource initialize to 1.
    - some change on data, this is a more constrained operation than read
                        // pthread_mutex_lock, pthread_mutex_unlock
                        // main sets the reader, writer threads
                        // main routine is also called test.driver
s=5;
// if we don't want to keep track of the s.prons and who is eating. otherwise initialize array.
                        // we will need mutex - for thread sync
                        writeblock - for resource sync
                        // we don't need a readblock because it is not exclusive in nature and it need not be restricted.

void * rdr(void * arg)
{
    int f = (int *)(arg);   // local reader no. initialized.
                            // reader no, writer no. to differentiate
                                R1, R2, R3, ...
                                W1, W2, W3, ...
                            thread no.
                            input is usually 'i'.
    sem_wait(&mutex);
    rcount=rcount+1;        // first read is allowed.
                            // no more reads ≡ rcount=0
                            else rcount>0.
    if(rcount==1)           // first read
                            // will be blocked if lock is with someone else
        sem_wait(&writeblock);
    sem_post(&mutex);       // mutex is unlocked
                            // suspend the write until the read is complete. i.e. block write
                            // otherwise 2nd read/write input will be allowed (2) (3) will be allowed
    printf("%d %d", data, f);
    sleep(5);               // or any no.   // simulating context switch from one thread to other threads in the queue (1) i.e. to relinquish control.
                            simulate other person coming in (2)
    sem_wait(&mutex);
    rcount--;               // as we are touching shared rcount variable
    if(rcount==0)           // nobody is reading data.
        sem_post(&writeblock);
    sem_post(&mutex);
```

```
void * writer wtr(void * arg)
{
    f = arg;
    sem_wait(&writeblock);   ——— access writeblock so that we can change data.
    data++;
    printf("... %d %d", f, data);   ——— thread no.
    sleep(2);   ——— to allow other threads to busy wait (?)
    sem_post(writeblock);
}
main()
{
    pthread_t rtid[5], wtid[5];   ——— any no. of threads.
    for(i=0; i<5; i++)
    {
        pthread_create(&wtid[i], NULL, wtr, i);   ——— (void*) thread index
        pthread_create(&rtid[i], NULL, rdr, i);
    }
    for(i=0; i<5; i++)
    {
        pthread_join(wtid[i], NULL);
        pthread_join(rtid[i], NULL);
    }
}
```

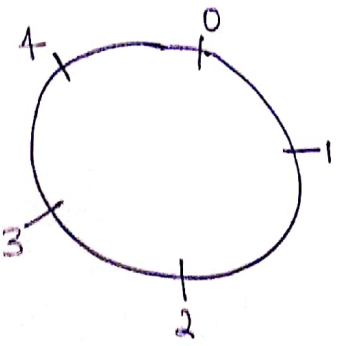
scope issues.

initialized resource will be accessed in a shared manner as global variable

resource count

sem_init(&mutex, 0, 1);

⟹ Dining philosopher's problem.

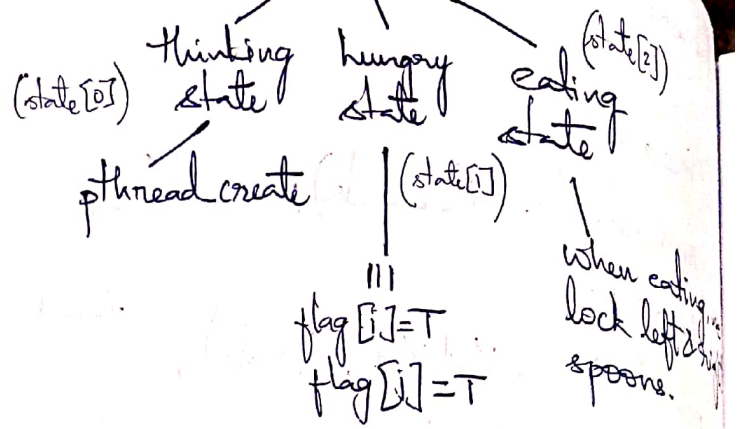


P, Q...

t_1	t_2
sem_wait(P);	sem_wait(Q);
sem_wait(Q);	sem_wait(P);
⋮	
sem_post(P);	sem_post(Q);
sem_post(Q);	sem_post(P); (?)

This is a deadlock code.

Every philosopher has 3 states T H E




```
#define N 5;
#define T 0;
#define H 1;
#define E 2;
int state[ph_num];   // 0-4
sem_t S[N];
```

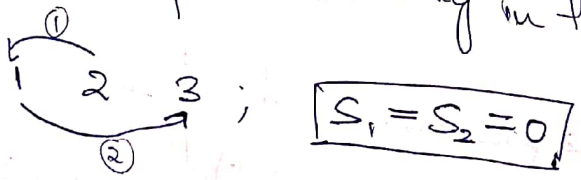
28/10/19.

SEMAPHORES numericals.

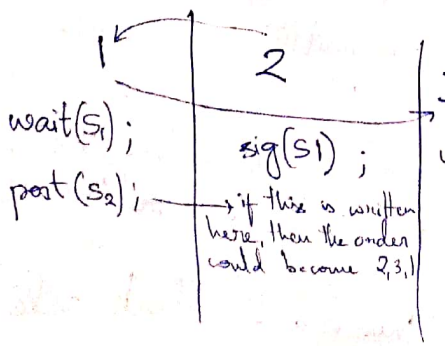
need for semaphores - when resources are multiple in nature, we use it for synchronization.

CSP — no race ☆

eg: We have 3 process executing in the given order



* signal = post



eg. There are 2 processes P & Q

```
P                    Q
while(1){            while(1){
  W;                   Y;
  P("0");              P("1");
  P("0");              P("1");
  X                    Z;
```

X, Y, Z, W are instruct^{n}s
output should be
00110011 (PQPQ)

Given V → SIGNAL = post
P → wait

two resources are S & T

	W	X	Y	Z	S	T
~~(I)~~	P(S)	V(S)	P(T)	V(T)	1	1
✓(II)	P(S)	V(T)	P(T)	V(S)	1	0
~~(III)~~	P(S)	V(T)	P(T)	V(S)	1	1
~~(IV)~~	P(S)	V(S)	P(T)	V(T)	1	0

(I) — P & Q can parallely get into CS, would give 1100... so wrong.
(II) — strict alternation bet^n P & Q.
(III) — non-satisfact^n of order we want.
(IV) — not releasing or unlocking for Q to get in.

HW eg What should be the ordering of W, X, Y, Z, such that $01^n0/10^n1$ ~~should~~ not be generated and given that 'n' is odd.

010X
101X

	W	X	Y	Z	S	T
(I)	P(S)	V(S)	P(T)	V(T)	1	1
(II)	P(S)	V(T)	P(T)	V(S)	1	1
(III)	P(S)	V(S)	P(S)	V(S)	1	X
(IV)	V(S)	V(T)	P(S)	P(T)	1	1

(III) X — dont care.

while(s<=0);
s--;

S:0 1 0
S:0 2 1 0

eg.

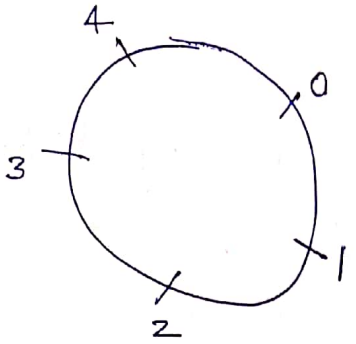
P_1	P_2
① $C = B-1$	$D = 2B$ ③
② $B = 2C$	$B = D-1$ ④

with starting with $B = 2$, how many different values of B in any order of execution.

29/10/19.

⇒ Synchronization problems.

→ Dining philosophers



→ Readers writers setup.

R(n)	W(1)
n reads can happen at the same time provided no write happens in between.	stops a write when read is going on and writes, when rcount = 0

crit. data, rcount = 0

sem_t mutex, wrt;

(initialize 1) (initialize 1)

* intra thread synchronization
- five threads and every thread gets an opportunity

Dining philosophers

```
#define N 5
#define T 0
#define H 1
#define E 2
int state[N];
phil_num[N] = {0,1,2,3,4}; // can have from 0-4 or 1-5
sem_t mutex; S[N]; // from above
```

T	0
H	1
E	2

"flag"

5-philosophers, so no size of array = 5 * threads.

* notion of datatype has is only a string of macro (i.e. # define) Or we could also use enumeration.

```
#define LEFT (ph_num + 4) % N
#define RIGHT (ph_num + 1) % N

void * phil(void * num);
void take_fork(int );
void put_fork(int );
void test(int * ); // can a philosopher go from H to E state i.e. can not
                   // the neighboring phil. eating.
test(LEFT)
int main()
{
  pthread_t tid[N];
  sem_init(&m, 0, 1);
  sem_init(&s, 0, 1);
  for(i=0; i<N; i++)
  {
    sem_init(&s[i], 0, 0);
  }
  for(i=0; i<N; i++)
  {
    pthread_create(&tid[i], N, phil, &phil_num[i]);
    printf("philosopher is thinking %d", i+1);
  }
}
```

// we need not set the thinking state initially, but after sometime, we have to change the eating state to thinking state which needs to be mentioned.

later we should increment locked presently

to map 0—4 to 1—5

```
void * phil(void * num)
{
  while(1)
  {
    int * i = num;
    tf(*i); // take fork  → sleep(5); // sleep so that we can see
    pt(*i); // put fork              the philosopher eating and
    (1) sleep(5);                    the next waiting.
  }
}

void takeforks(int ph_num)
{
  sem_wait(& mutex).
  state[ph_num] = H;
  printf("phil num H %d", ph_num+1);
  test(ph_num); // check if we can change phil state from H->E
  sem_post(& mutex);
  sem_post(&s[ph_num]);
}

void test(ph_num)
{
  sem
  if ((state[ph_num] == H) && (state[L] != E) && (state[R] != E))
  {
    state[ph_num] = E;
    printf("eating %d", ph_num+1);
  }
  ← sem_wait(& s[ph_num]);
}
```

4/11/19

Deadlock — Detection → (Identification) → DAG → (precedence)
Deadlock — Prevention
Deadlock — Avoidance

Lock based solutions are for preventing deadlocks or for data consistency.

→ Process or threads are holding resources that are required for the other processes

→ Deadlock in dining philosophers — when every philosopher picks his left spoon.

→ Deadlock detection - to check if a system is in deadlock state or no.

→ Characterize the features associated with deadlock (it is an and condition)

1) Mutex
2) Hold and wait
3) No preemption (from resource pt. of view)
4) Cyclic / circular wait

→ Detection - the system checks if will get pushed to deadlock state or not is in

Resolution:

→ Linux will not automatically support deadlock detection and all that because pratically it takes a lot of time.

→ The best way for avoidance

Soft way of handling → close the power.

Hard way of handling → pull the plug

→ Avoidance - looks at all possible future cases

Prevention — t=0 t=1

look for all possible sequences and check to see when we can move from one safe state to another safe state.

But since there are many combinations, it is considered to be impractical (eg banker's trace)

* Here unsafe state is deadlock state.
* tracing is a sort of empirical analysis.
* all the four conditions stated above should be satisfied for the deadlock to occur.

i.e. ~~1 & 2~~ $a \wedge b \wedge c \wedge d$

$$\neg(a \wedge b \wedge c \wedge d) = \bar{a} \vee \bar{b} \vee \bar{c} \vee \bar{d}$$

i.e. even if one of them is not satisfied, then deadlock ~~doesn't occur~~ can be prevented from happening in further states.

1) Mutex - the system can be in a deadlock state if there is atleast ~~any~~ one of the resources is accessible in a non-sharing manner

* Interleaved vs ~~conco~~ concurrent

mostly in a uniprocess setup	mostly in a multi-process setup.

* Atleast one should be in a mutex setup

2) Hold & wait - a process will be holding a resource required by another process and it waits for another resource which some other process is holding. Every process in a deadlock state will satisfy both hold and wait.

In both scenarios of only hold or only wait is not a deadlock.

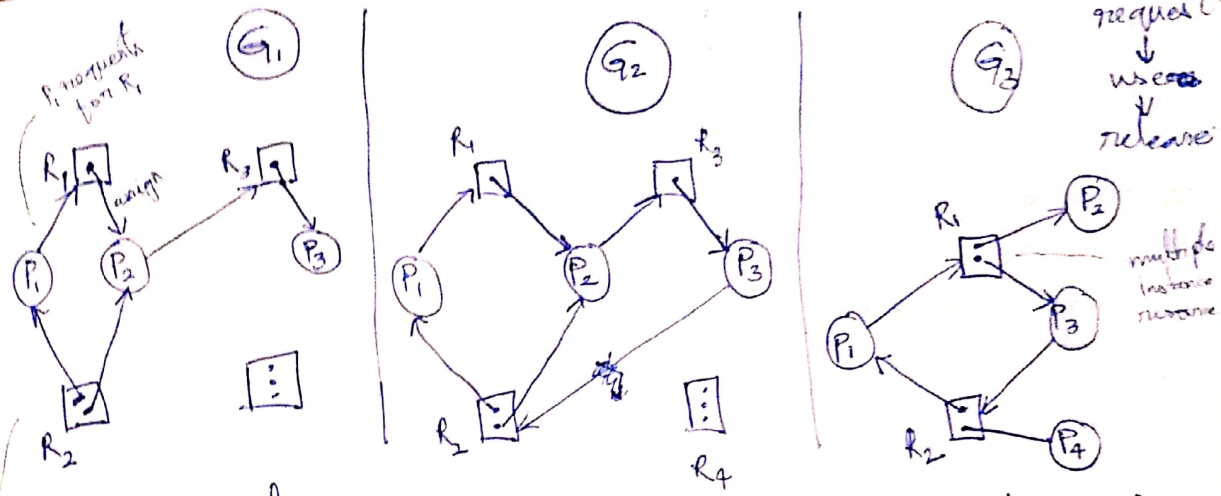
3) No preemption - no preemption of resources from ~~the~~ processes

* rmdir - removes only when subdir. are empty.

forceful preemption of resources is not allowed.

Preemption should be blocked for deadlocks to occur as resources can be removed from the process otherwise.

* Negative and positive weibull.



4) Cyclic or circular wait — this is the better way of negating and preventing deadlock.

→ RAG — resource allocation graph

Process and ~~resources~~ resources are both are represented using vertices.

edges → for all kinds of connections.

↳ resource allocated to ~~the~~ process or - - - - - -

• → how many copies of resources are there.

→ After graph construction, try if you can create atleast one cycle

G_1 — devoid of cycles — deadlock free setup.

* Checking for cycles is necc and suff. condition for deadlock unless there is no multiple instance resource involved in cycle creation. ~~Otherwise~~ Otherwise it is just necessary but not sufficient.

* If we are interested in resolution, we will need to have the vertices of the cycle inorder to break the deadlock.

* kill P_3 ≡ breaking cycle ≡ resource preemption ≡ ¬C ≡ deadlock won't occur.

G_2 — has deadlock (when P_3 is not killed)

G_3 — ~~has deadlock~~ the cycle is present ~~by~~ but R_2 is a multiple instance resource involved and is being accessed by another process not involved in ~~deadlock~~ cycle. Therefore, it is not a deadlock as P_4 will release R_2 and then P_3 can access R_2

⇒ Prevention: $(\neg a \vee \neg b \vee \neg c \vee \neg d)$

$\neg a$ X as printer can't be shared by 2 threads or processes

violation of b — not all process are in hold & wait state

- some process never in wait state. i.e. the process will get all its resources before it proceeds.
- ✱ Not a feasible one to violate

$\neg$ hold ≡ a process has 3R and needs 2R, so it just releases the 3R it has so that atleast some other process can use it.

Even if one resource is being waited for, then it will relinquish the resource it has but it will lead to starvation and it can never go to its CS.

⇒ Cyclic wait

cyclic wait ~~should not be there for~~ → deadlock

F(R): R → N (mapped to)

- R: resource name
- N: real no.

F(tape_drive) = 1, F(printer) = 5, F(DVD) = 10

read (DVD) → paste (on HD) → print

A process will be allowed to submit a ~~resource~~ request only when it is a higher no.

i.e. printer can submit request for DVD and not for tape_drive.

If we have R_j and R_i

then R_i can ~~only~~ submit request only if $F(R_j) > F(R_i)$

⇒ Deadlock avoidance — Banker's algo

(Safe state identification)

$\neg$ (mutex & hold_wait & no preem. & circular_wait)

preemption allowed — (to avoid deadlock)

- wait-die
- wound-wait

wait-die:

wound-wait: kill others and wait for myself

P_i has R_1, R_2

but wants R_3 which is with P_j

if R_3 is not available after a particular time quantum, P_i R_2 are released by P_i in wait_die.

But in wound_wait, if P_i requires R_3, R_4, then P_i wounds P_j, or kills P_j to obtain R_3, R_4.

Killing is based on arrival time or time that process had on the CPU.

Both the above are feasible to depict preemption.

* sudo - super user do (?)

→ Circular wait should be satisfied

Assume no circular wait

$F(R) \longrightarrow N$

P_i has R_i and waits for R_j

$F(R_j) > F(R_i)$ → * this is referred to as total ordering of resources.

If we have total ordering in place, there won't be deadlock

eg.

P_1	P_2
	R_2
$W(R_1)$	
$W(R_2)$	

* we have to relinquish R_1 as $1 < 2$, so there won't be deadlock.

* $P_{i+1}(R_i) \longrightarrow P_i$

$F(R_i) < F(R_{i+1})$

i.e. $F(R_0) < F(R_1)$ and so on.

by transitivity, there won't be cyclic wait.

eg.

claim matrix

allocation matrix.

	Max	Need
P_0	10	5
P_1	4	2
P_2	9	2
		9

A resource R of 12 copies allowed.

~~max allocation — more than that n worst possible requests~~

start with 1 → 0 → 2

12−9 = 3 copies of resources left — available vector V = 3

P_1 currently has 2 Rs

* max claim should never exceed resource count.

* assuming all processes run till completion

* Banker's algo — Stallings.

1 → 0 → 2

There are 3 R
1 requests for 4−2 = 2R,
so available are 3−2 = 1R.
After 1 is complete, there would be 1R+4R = 5R

There are 5R
0 requests for 5R,
so available = 0R
After 0 is complete, there are 5R+10R = 10R

There are 10R available
2 requests for 7R,
so available = 10−7 = 3R.
After 2 is complete,
there are 3R+9R = 12R

∴ 1, 0, 2 is a safe sequence.

⟹ Deadlock avoidance/banker's algo.

* As in practical applications, there are many resources taken into consideration, so avoidance is not feasible.
* Approach for prevention and avoidance is different
* formal methods avoid empirical approach (?)
* safety critical — keep the percentage of failure low.
* testing — ~~prove~~ show that the software works for all conditions
* prevention is the mathematical/formal approach to avoid deadlock

* prevention is less flexible than avoidance.

eg. P_2 asks for 1 more R_1 using the same table above. There is a potential for deadlock.

2,1,3 is an unsafe state

* Deadlock - process thrashing.

Claim (C)

	R_1	R_2	R_3
P_1	3	2	2
P_2	6	1	3
P_3	3	1	4
P_4	4	2	2

Allocation (A)

	1	0	0
	6	1	2
	2	1	1
	0	0	2
V =	0	1	1

C-A

2	2	2
0	0	1
1	0	3
4	2	0

Resource vector = | 9 | 3 | 6 |

↓

We can meet requirement only if C-A < V

3/11/19

(Virtual)

→ Memory management → Segmentation
→ Paging
→ Fitting, page replacement, algos (swap)

2 strategies
- segmentation
- paging.

Fitting — processes will be fitted into which memory block.

[logical memory] — memory which doesn't exist ..... ①
exists on soft copy

* memory — collection of bytes.

* Physical memory - real time addresses.

* We should somehow efficiently map logical memory and physical memory.

The application only interacts with logic memory. This logic is then mapped to physical memory.

* Mapping to non-existing memory — initially program is loaded to logical memory — then to physical memory.
It is because RAM is scarce and costly. We can't load every possible ~~binaries~~ binary.
Things are loaded only when they are called/ ~~used~~ needed.
* Web server — Debian linux.

⟹ Solution for last class's example.

⟹ 3 2 2 1 0 0

6 1 3 5 1 1 ~~or 3 1 1~~

3 1 4 2 1 1

4 2 2 0 0 2 V = 1, 1, 2

i) $P_2 \xrightarrow{\text{submits requests}}$ 1 more of (R_1, R_3)

ii) $P_1 \xrightarrow{\text{submits request}}$ 1 more of (R_1, R_3)

→ Segmentation, fitting strategies

- first fit
- best fit
- worst fit

CPU addrss = logical addrss.

* CPU addresses are larger in size

physical addrss = absolute addrss

* logical memory ≡ simulated/software memory.
* dynamic link libraries – get linked to the code at runtime.
* stubs – empty slots
* memory view
* programmers point of view about memory ≡ symbolic addresses

compiler → relocatable addrss.

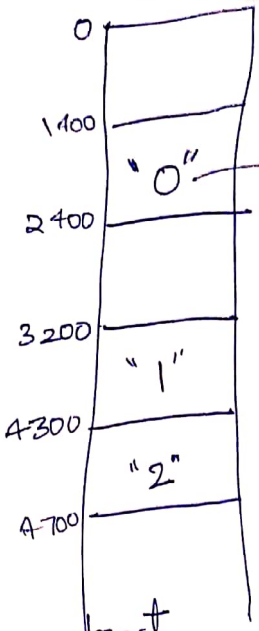
finally mapped into → absolute addrss.

* data and functions should be stored in the memory. # The mapping helps here.
* Segmentation & paging : mapping 2D array to 1D
* memory —
  - heap segment
  - global variable segment
  - code segment
  - library segment
  - stack segment.
* user mode, privilege mode
* start addrss, base addrss, limit

Limit

→ Segment table

	Limit	Base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700



where we have the liberty of mapping a segment at the place we want/wish

If there is an attempt to extract (2,53), then it would be 4300+53 = 4353.

* Segment is a one level better that contiguous allotment
* equipartitioning scheme/fixed - all partitions are of equal size.

→ collection of address ≡ segment.

→ left over places in the memory which can't even be allot allocated to some process (as it is not in a contiguous fashion) are called fragments.

→ holes ≡ unused spaces in memory

→ holes quelescing - merging the holes

→ Paging - memory management soln.

↳ NCA requirement solved.