Concurrency #6:

→ Finish sema (P/C problem)

→ Deadlock

```c
int max;
int loops;
int *buffer;

int use  = 0;
int fill = 0;

#define CMAX (10)
int consumers = 1;

void do_fill(int value)
{
    buffer[fill] = value;
    fill++;
    if (fill == max)
        fill = 0;
}

int do_get()
{
    int tmp = buffer[use];
    use++;
    if (use == max)
        use = 0;
    return tmp;
}

void *
producer(void *arg)
{
    int i;
    for (i = 0; i < loops; i++) {
        do_fill(i);
    }
}

void *
consumer(void *arg)
{
    int tmp = 0;
    while (tmp != -1) {
        tmp = do_get();
        printf("%d\n", tmp);
    }
}
```

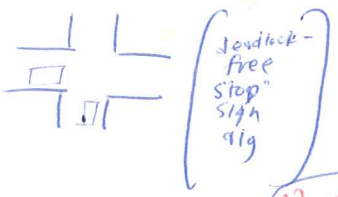
Deadlock

everyone waiting for something held by other
none release until they get what they're waiting for

A ⇄ B

A → B
↑ ↓
D ← C

e.g. stop sign



(deadlock-free "stop" sign alg)

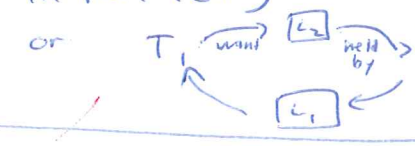
1st Kansas: "when two trains approach each other @ crossing, both shall stop and neither start til other has gone"

⇒ code example:

T_1: sem-wait(lock1); sem-wait(lock2);
T_2: sem-wait(lock2); sem-wait(lock1);

(happens in real code)

$T_1 \rightleftarrows T_2$ or



talk about: encapsulation

Java example: V_1.addAll(V_2) ; V_2.addAll(V_1)

page fault VM, FS → VM

3) Detect/Recover

⇒ detect: keep track of waits-for graph
if cycle detected,
→ tell human?
→ restore thread/system
(DBMS)

challenge: not leaving system in funny state

(lock 1
x = subtract from acct 1
abort here → lock 2
add x to acct 2
unlock 2
unlock 1)

Conditions

mutual exclusion: ≥1 resource held in non-shareable mode; another req → delay
hold-and-wait: ∃ process holding, may request new, waiting
no preemption: can't be preempted
circular wait: ∃ $P_0, \ldots P_n$ s.t. $P_0 \to P_1 \to \ldots \to P_N \to P_0$

solutions

prevent: ensure 1/more conditions doesn't hold
~~avoid~~ ⇒ scheduling
detect/recover: allow but notice + recover
ignore: not everything worth doing is worth doing well

1) Prevent

→ mutex: (maybe read-only?) hard to get around this → lock-free structs
→ hold+wait: guarantee all locks grabbed @ once, atomically
(but encapsulation makes this hard)
→ also, starvation possible?

acquire
sem-wait(lock)
=
sem-post(lock)

→ no preemption:
mutex_trylock( )
⇒ may fail

→ circular wait
impose total order
always grab $lock_1$ before $lock_2$
(common approach)

wrote 2-lock acquire that works even if done in wrong order ⇒ (how to do w/ encapsulation?)

2) Avoidance:

negatives:
w/ smart scheduler
if you knew something about maximal possible requests of a thread, could decide how to schedule
(Banker's alg, others)

e.g. T_1 begin L_1 L_2 exit; T_2 begin L_2 L_1 exit; T_3 begin L_3 exit; T_4 no locks

wait-free synch.

insert_at_head (ins):

```
node_t *new = malloc( );
new->element = element;
new->next = head;
head = new;
```

⇒ is this a critical section?

(example)

```
int CAS (unsigned *addr, unsigned exp, unsigned val)
if (*addr == exp) {
    *addr = val
    ret 1
}
ret 0;
```

⇒ how to rewrite list insert s.t. it uses CAS to update list w/o locks?

```
node_t *new = malloc( );
do {
    node_t *old = head;
    new->next = head;
} while (CAS(&head, old, new) == 0);
```