Synch

Prof. K. Gopinath
IISc

Fundamentals

- P1: y=y+1 || P2: y=y-1 ; initially y=1
  - Interleaving model of concurrency
    - Only result: y=1
  - True concurrency model
    - y can be 0,1,2!
      - 0: t=y; s=y-1; y=t+1; y=s
      - 1: P1 atomically followed by P2 or vice versa
      - 2: t=y; s=y-1; y=s; y=t+1;
  - Granularity of atomic actions true reason for diff
- P1: t=y; y=t+1 || P2: s=y-1; y=s
  - Only one "critical" reference
  - y can be 0,1,2

Synch

- Two types of synch:
  - Mutual Exclusion
  - Condition synchronization
- Fine-grained synch: using HW primitives
- Coarse-grained synch: constructed atomic actions
- Properties:
  - Mutual Exclusion (safety)
  - Absence of deadlock or progress (liveness)
  - Absence of unnecessary delay (safety)
    - Should depend only on processes trying to enter
  - Eventual Entry (liveness)
  - Sometimes also worry about “bounded wait”

Terms

- Atomic actions; Critical sections; history; trace
- Deadlock: no transitions to other states
- Livelock: “busy loop”: transitions to other states but does not change anything
- Safety: nothing bad happens
  - Eg: partial correctness
- Liveness: something good happens eventually
  - Eg: termination;
- Every property can be formulated in terms of safety and liveness properties
  - total correctness: safety+liveness

Scheduling Policy: Fairness

- Unconditional Fairness: every unconditional atomic action is eventually executed eg: round robin; Peterson's alg/tie-breaker alg
- Weakly fair: (1) unconditionally fair (2) every conditional atomic action that is eligible executed eventually, assuming that its guard becomes true and is not subsequently falsified, except possibly by the process executing the conditional atomic action
- strongly fair: (1) unconditionally fair (2) every conditional atomic action that is eligible executed eventually, assuming that its guard is infinitely often true eg: test&set

```
      continue=true; try=false;
Loop: while(continue) {
          try=true;
          try=false
       }
Stop: <await try --> continue=false>
```

weak fairness: may not terminate

strong fairness: will terminate
    eventually

Mutual Exclusion

```
P: while{
      Entry protocol
      Critical section
      Exit protocol
      Non-critical section
   }    =>
bool in1=in2=false;
// Mutex: ~(in1 && in2)
P1: while(1) {
       in1=true;
       <Critical section;>
       in1=false;
       <Non-critical section;>
    }
P2: ...
```

```
bool in1=in2=false;
// Mutex: ~(in1 && in2)
P1: while(1) { // Mutex && ~in1
          <await ~in2 --> in1=true;>
          // Mutex && in1
          <Critical section;>
          in1=false;
          <Non-critical section;>
          // Mutex && ~in1
       }
P2: ...
<await ~in2 --> in1=true;>:
     while (in2); in1=true  ? no Mutex
     in1=true; while (in2)  ? deadlock
<await !lock --> lock=true>
```

Test & Set

TS(lock, cc)
<cc=lock; lock=true>

- Lock:
  - Repeat
  -     TS(lock, cc)
  - Until !cc
- Unlock:
  - lock=false
- Too much contention for the bus!

Test & Test & Set

- Lock:
  - Repeat
  -     while (lock);
  -     TS(lock, cc)
  - Until !cc

Peterson's alg

- Break deadlock by a tie: let last be the one that entered cs last if both interested

```
bool in1=in2=false;
// Mutex: ~(in1 && in2)
P1: while(1) { // Mutex && ~in1
        in1=true; last=1;
        <await ~in2 or last=2>
        // Mutex && in1
        <Critical section;>
        in1=false;
        <Non-critical section;>
        // Mutex && ~in1
     }
```

```
bool in1=in2=false;
// Mutex: ~(in1 && in2)
P1: while(1) { // Mutex && ~in1
        in1=true; last=1;
        while(in2 && last=1);
         // Mutex && in1
        <Critical section;>
        in1=false;
        <Non-critical section;>
        // Mutex && ~in1
     }
```

Both P1 and P2

   read/write last (multi-writer)

   write their own in but read other's

Subtlety in Peterson's alg

Note that if last=1; followed by in1=true;

INCORRECT! COMPILER OPTS!

```
P1: 0 while(1) {
      1   <Non- Critical section;>
      2   last=1;
      3   in1=true;
      4   while(in2 && last=1);
      5   <Critical section;>
      6   in1=false;
    }
```

Let state of 2 processes be

(PC1, PC2, last, in1, in2)

(PC1, PC2, last, in1, in2)	[last,in1/2]
(0, 0, 1, 0, 0)	
(1, 0, 1, 0, 0)	
(1, 1, 1, 0, 0)	
(2, 1, 1, 0, 0)	
(2, 2, 1, 0, 0)	
(3, 2, 1, 0, 0)	[1,1,0]
(3, 3, 2, 0, 0)	[1,1,1]
(3, 4, 2, 0, 1)	[2,1,1]
(3, 5, 2, 0, 1)	PC2 blocked
(4, 5, 2, 1, 1) !!!	
(5, 5, 2, 1, 1)	
INCORRECT code	CORRECT

2-Process Solution (Peterson '81)

```
process p ::

initially ¬ flag[p] ∧ p@ 2

while true do
   2: NCS;
   1: flag[p] := true;
   0: turn := p;
   5: await (¬flag[q] ∨ turn = q);
   4: CS;
   3: flag[p] := false
od
```

```
process q ::

initially ¬ flag[q] ∧ q@ 2

while true do
   2: NCS;
   1: flag[q] := true;
   0: turn := q;
   5: await (¬flag[p]∨turn = p);
   4: CS;
   3: flag[q] := false
od
```

Exclusion Property

- Exclusion is a **safety** property (indicates that something bad does not happen).

$$\neg(\ p@4 \wedge q@4)$$

- How to prove safety properties? **Invariants**.

- An assertion is an invariant if it is **true in every state every time the program runs** (*i.e.*, it is **always** true).

How to Prove an Invariant

- **Use known invariants:**

$$\frac{\text{invariant } I, \quad I \Rightarrow J}{\text{invariant } J}$$

- **By induction:**

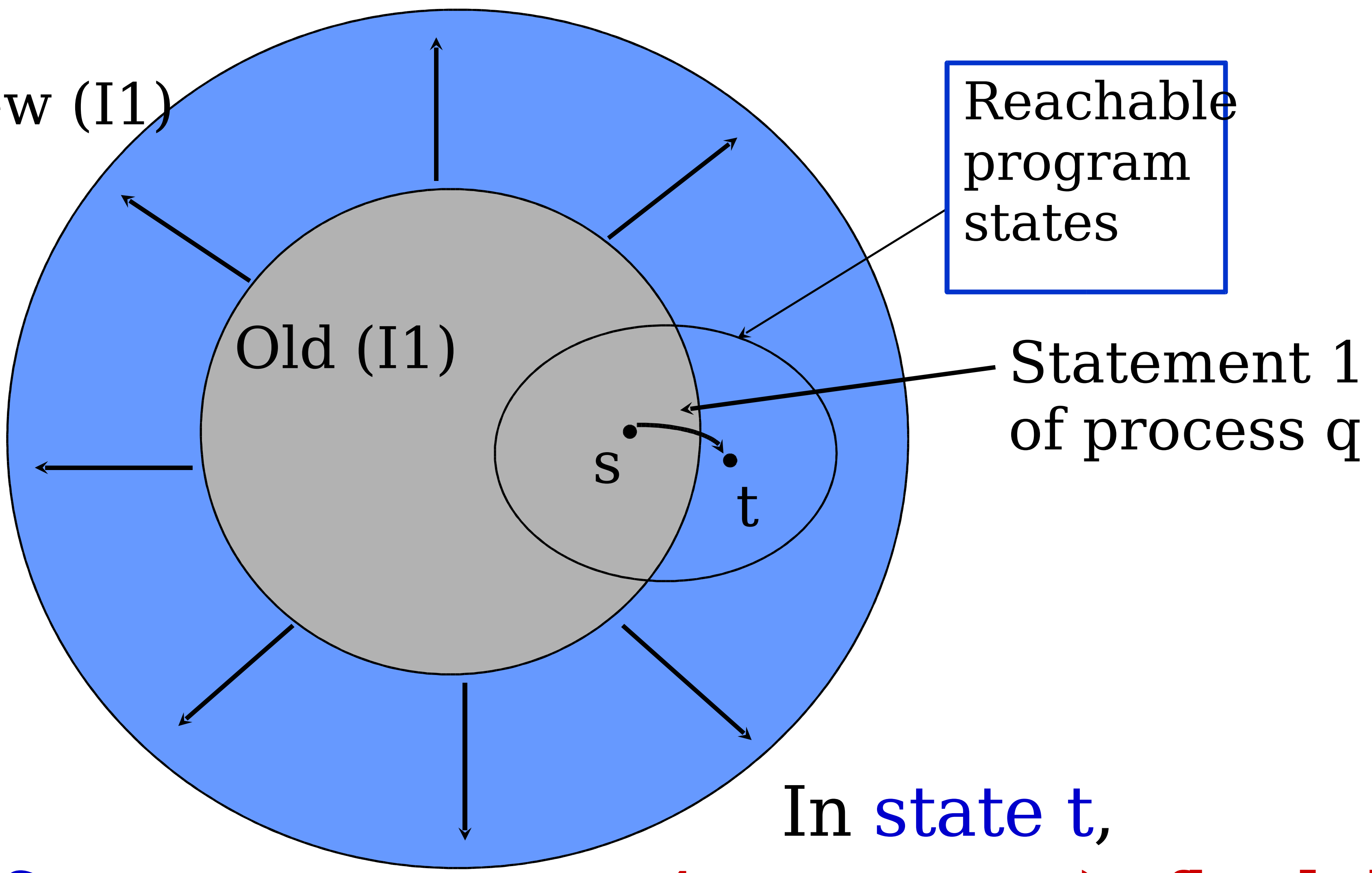
$$\frac{\begin{array}{l}\textbf{(i)}\ I \text{ holds initially} \\ \textbf{(ii)}\ I \text{ not falsified by} \\ \qquad \text{any statement} \\ \qquad \text{execution}\end{array}}{\text{invariant } I}$$

Proof of Exclusion Property

- **Objective:** Prove $\neg(p@4 \wedge q@4)$ (I0)
- **Strategy:** Derive other invariants that imply (I0)
- Define an assertion that describes program's "state" when *p* is in CS.
- **1st attempt:** $p@4 \Rightarrow \neg\, flag[q] \vee (turn = q)$(I1)

  (from await condition)
- Is this an invariant? Initially true, not falsified by statements of *p*, **but** falsified by statement 1 of *q* when $p@4 \wedge (turn = p)$ holds
- Potential invariant is too strong: **Weaken it.**

What Does "Weaken It" Mean?



In state t,
p@4 ∧(turn = p)∧ flag[q]
holds,so (I1) is violated.

In state s,
p@4 ∧(turn = p) holds

Try a Weaker Invariant

- Weaken invariant by adding extra disjunct to the consequent.

$$p@4 \Rightarrow \neg\, flag[q] \vee (turn = q) \vee q@0 \qquad (I1)$$

- Is this an invariant? **Yes!!!**
- Similarly, the following is an invariant

$$q@4 \Rightarrow \neg\, flag[p] \vee (turn = p) \vee p@0 \qquad (I2)$$

Exclusion Proof (Continued)

- Two more simple invariants are required:

  $p@\{0,3..5\} \Rightarrow flag[p]$ (I3)

  $q@\{0,3..5\} \Rightarrow flag[q]$ (I4)

- Finally, we want to prove that (I1), (I2), (I3), and (I4) together imply (I0).

♦ **Strategy:** Use (I1) through (I4) to prove:

  $p@4 \land q@4 \Rightarrow false$ (which implies (I0))

Proof of (I0)

$p@4 \wedge q@4$

$\Rightarrow p@4 \wedge q@4 \wedge flag[p] \wedge flag[q]$, by (I3) and (I4)

$\Rightarrow (turn = p) \wedge (turn = q)$, by (I1) and (I2)

$\Rightarrow false$, predicate calculus

This concludes the proof of (I0).

Progress Proof

- Progress is a **liveness** property (indicates that something good eventually happens).
- Progress properties are proved using "leads-to" assertions.

♦ *A* leads-to *B* iff whenever *A* holds, either *B* holds, or *B* will hold in the future.

- How can we prove leads-to assertions?

Proving *A* leads-to *B*

1) From other leads-to assertions (*e.g.*, using transitivity). **Example:**

$$\frac{A \text{ leads-to } C,\ C \text{ leads-to } B}{A \text{ leads-to } B}$$

2) From program text and fairness.

Example:

```
   ⋮
4: x := 0;              p@4  leads-to  p@5
5: y := 1;
   ⋮
```

Proving *A* leads-to *B*

3) Use a **well-founded ranking**.

$$\boxed{R: S \rightarrow T}$$

S: the states of the program

T: an ordered set with no infinite chains such that

$$t_0 > t_1 > t_2 > \ldots\ldots > t_k > \ldots$$

Example: Non-negative integers.

Progress Proof

- First, we need the following invariant:

$$
\begin{aligned}
& p@\{2,5\} \wedge q@\{2,5\} \Rightarrow \\
& (\, p@5 \wedge (\neg flag[q] \vee turn = q)) \vee \\
& (\, q@5 \wedge (\neg flag[p] \vee turn = p)) \vee \\
& (\, p@2 \wedge q@2)) \qquad (I5)
\end{aligned}
$$

- This assertion shows that, unless both processes are in their nonCS, some statement is enabled. That is, there is no **deadlock**.

Progress Proof

- To prove progress for *p*, prove

  p@1 leads-to p@4 (L0)

  (progress for exit section is trivial).

- Proof overview:

  p@1 leads-to p@5, prog. text and fairness

  p@5 leads-to p@4 (L1)

♦ (L0) follows from above and transitivity.

Proof of (L1)

- Clearly, it suffices to prove

  p@5 leads-to ¬p@5.

- Define well-founded ranking *R* as follows:

  (**Note:** Could use “prog. text and fairness” instead.)

$$R = \begin{cases} 0 & \text{if } \neg p@5 \\ 1 & \text{if } p@5 \wedge turn{=}q \wedge q@5 \\ q.pc+2 & \text{if } p@5 \wedge (turn{=}p \vee \neg q@5) \end{cases}$$

(*q.pc* “ program counter of process *q*”)

Proof of (L1)

1) $p@5 \Rightarrow R = 1 \vee R = q.pc+2$
$\Rightarrow R > 0$

2) $R=0 \Rightarrow \neg p@5$

We show that, if $R > 0$, then **each** statement execution decreases R. (Note that (I5) implies that while $p@5$ holds, there is always an enabled statement.)

First, check statements of process p. $R > 0$ implies $p@5$. Statement 5 decreases R to zero.

Proof of (L1)

Check statements of process q.

Statements 1, 2, 3, & 4:

$R = q.pc + 2$ before these statements are executed. These statements do not affect *turn* or $q@5$, but do decrease $q.pc$. Therefore, they all decrease R.

Statement 0:

If R > 0 holds before, then $p@5 \wedge q@0$ holds, so $R = 2$. After execution, $p@5 \wedge turn = q \wedge q@5$ holdsTh erefore, $R = 1$ holds afterwards.

Proof of (L1)

Statement 5:

If $R > 0$ before, then

$p@5 \wedge q@5$

$\Rightarrow p@5 \wedge q@5 \wedge flag[p],$ by (I3)

$\Rightarrow p@5 \wedge q@5 \wedge (turn=p),$

statement 5 of q is enabled

$\Rightarrow R=5+2=7$

After execution, $p@5 \wedge q@4 \wedge (turn=p)$, so

$R = 4 + 2 = 6$. This concludes proof of (L0).

n-process solution

- Use 2-process solution as basis
- Entry protocol loops thru n-1 stages
- Each stage uses a 2-process solution to determine winner
- in[i] indicates which stage P[i] is executing
- last[j] indicates which process was last to begin stage j
- Atmost n-i processes past ith stage
- Solution is livelock-free, avoids unnecessary delay, ensures eventual entry
- O(n^2)
- Bounded numbers
  - No large #
- Note perf bug; should be for j = 1 to n-1

```
int in[1:n] = ([n] 0), last[1:n] = ([n] 0);
process CS[i = 1 to n] {
  while (true) {
    for [j = 1 to n] {          /* entry protocol */
      /* remember process i is in stage j and is last */
      last[j] = i; in[i] = j;
      for [k = 1 to n st i != k] {
        /* wait if process k is in higher numbered stage
           and process i was the last to enter stage j */
        while (in[k] >= in[i] and last[j] == i) skip;
      }
    }
    critical section;
    in[i] = 0;                  /* exit protocol */
    noncritical section;
  }
}
```

Figure 3.7 The n-process tie-breaker algorithm.

Ticket Alg

- Get a ticket # larger than any previous; wait till ticket # is next
- TICKET Invariant: P[i] in cs =>
  - turn[i]=next &
  - All non-0 values of turn unique: for all i, j: 1<=i, j<=n, j != i: turn[i]=0 or turn[i] != turn[j]
- Bakery algorithm: instead of above ticket alg, sets its number 1 larger than any existing & waits till it is smallest
- BAKERY Invariant: P[i] in cs => turn[i] !=0 & for all j: 1<=j<=n, j!=i: turn[j]=0 or turn[i]<turn[j]
- Problem: unbounded numbers (next, number)
- Need fetch&add for a fine grained solution
- FA(var, incr):
  <temp=var; var+:=incr; return temp>
- Also thru test&set
  - Critical section entry
  - turn[i]=number
  - Number+=1
  - Critical section exit

```
int number = 1, next = 1, turn[1:n] = ([n] 0);
## predicate TICKET is a global invariant (see text)
process CS[i = 1 to n] {
  while (true) {
    ⟨turn[i] = number; number = number + 1;⟩
    ⟨await (turn[i] == next);⟩
    critical section;
    ⟨next = next + 1;⟩
    noncritical section;
  }
}
```

Figure 3.8 The ticket algorithm: Coarse-grained solution.

```
int number = 1, next = 1, turn[1:n] = ([n] 0);
process CS[i = 1 to n] {
  while (true) {
    turn[i] = FA(number,1);          /* entry protocol */
    while (turn[i] != next) skip;
    critical section;
    next = next + 1;                 /* exit protocol  */
    noncritical section;
  }
}
```

Figure 3.9 The ticket algorithm: Fine-grained solution.

```
int turn[1:n] = ([n] 0);
## predicate BAKERY is a global invariant -- see text

process CS[i = 1 to n] {
  while (true) {
    ⟨turn[i] = max(turn[1:n]) + 1;⟩
    for [j = 1 to n st j != i]
      ⟨await (turn[j] == 0 or turn[i] < turn[j]);⟩
    critical section;
    turn[i] = 0;
    noncritical section;
  }
}
```

Figure 3.10 The bakery algorithm: Coarse-grained solution.

Fine Grained 2-process Bakery

```
turn1=turn2=0
P1:
while (1) {
   turn1=turn2+1
   while (turn2!=0 & turn1>turn2);
    critical section
    turn1=0
    non-critical section
}
P2:...
```

Prob: both P1 & P2 in c.s with init.

Soln: tie break: set condition to turn2>=turn1 in P2

Prob: race condition

P1 reads turn2=0 => cs

P2 reads turn1=0, sets turn2=1 => cs

Soln: each process sets turn$_i$ to *any value >=1*

```
while (1) {
   turn1=1; turn1=turn2+1
   while (turn2!=0&turn1>turn2);
    critical section
    turn1=0
    non-critical section
}
```

Symmetric Bakery

- P1 in cs: (turn1>0) & (turn2=0 or turn1<=turn2)
- P2 in cs: (turn2>0) & (turn1=0 or turn2<   turn1)
- Use lexicographic order to make it symmetric
  - turn1>turn2 in P1   --> (turn1,1) > (turn2,1)
  - turn2>=turn1 in P2 --> (turn2,2) > (turn1,1)
- n-process solution: Global turn[1:n] = 0
- BAKERY: P[i] in cs --> forall j: 1<=j<=n, j<>i: turn[j]=0 or turn[i]<turn[j] or (turn[i]=turn[j] & i<j)

```
P[i]: while (1) {
        turn[i]=1; turn[i]= max(turn[1..n])+1;
        forall j:1..n st i<>j  while ((turn[j]<>0) & (turn[i],i)>(turn[j], j));
        cs; turn[i]=0; non-cs
      }  // Note that there is no guarantee that 2 processes do not get same #
```

```
int turn[1:n] = ([n] 0);

process CS[i = 1 to n] {
  while (true) {
    turn[i] = 1; turn[i] = max(turn[1:n]) + 1;
    for [j = 1 to n st j != i]
      while (turn[j] != 0 and
               (turn[i],i) > (turn[j],j)) skip;
    critical section;
    turn[i] = 0;
    noncritical section;
  }
}
```

Figure 3.11 Bakery algorithm: Fine-grained solution.

Properties of Bakery

- requires that a process be able to read a word of memory while another process is writing it
  - variables read by multiple processes, but written by only a single process: good for dcs!
- bakery alg works regardless of what value is obtained by a read that overlaps a write
  - only the write must be performed correctly. The read may return any arbitrary value (a *safe* register)
  - implements mutual exclusion without relying on any lower-level mutual exclusion
  - <u>reading and writing need not be atomic ops</u>
  - Before this algorithm, it was believed that mutual exclusion problem unsolvable without using lower-level mutual exclusion (infinite regress!!!)

Lamport's Bakery alg for n processes

R

```
Process $P_i$
do {
T   D   $choosing[i] = true;$
        $number[i] = \max(number[0], number[1], \ldots number[n-1]) + 1;$
        $choosing[i] = false;$
    T-D $for(j = 0; j < n; j++) \{$
            $while(choosing[j]);$                                                        t1
            $while((number[j] != 0) \&\& ((number[j], j) < (number[i], i)));$           t2
        }
C           critical section                                                              s
E       $number[i] = 0;$
            remainder section
} while(1)
```

Structure of the algorithm

R - code prior to using Bakery algorithm

T - creating of a ticket and awaiting for permission to enter critical section

- D - creation of a number (first part of a ticket)
- T-D - awaiting for the permission to enter critical section

C - critical section itself

E - code executed upon exit from critical section

Basic Lemma

Lemma 1:

For any $i \neq j$, if P_i is in C and P_j is in C $\cup$ (T- D), then

$(number[i], i) < (number[j], j)$

Lemma 1 - Proof

- Consider time 's': P_i is in C (critical section)
  - number[i] has been selected and still exists
  - number[j] has been selected and still exists
  - Exists time point 't1'<'s', where P_i performs a check (choosing[j]==0) and passes it
  - Exists time point 't2', t1<t2<s, where P_i performs a check (number[j]!=0) and (number[i],i)<(number[j],j)
- Since at time ‘t1’ (choosing[j]==0), either
  - CASE A: number[j] was chosen after ‘t1’ but before ‘s’
  - CASE B: number[j] was chosen before ‘t1’

Lemma 1 – Proof – CASE A

- Since at time ‘t1’, P_i already checks for permission to enter critical section, computation of number[i] was computed before that and persists until ‘s’
- Thus, at the time P_j starts to compute its number[j], it has to take into account of ‘max’ value of number[i]. So it creates a value which is greater then number[i] at least by 1, and it persists until time ‘s’
- That is (number[i],i)<(number[j],j) at time ‘s’

Lemma 1 – Proof – CASE B

- Both number[i] and number[j] were computed before ‘t1’, thus also before time ‘t2’ and persisted until ‘s’
- At time ‘t2’, P_i performed check (number[j]!=0) & (number[j],j)<(number[i],i), which failed, since P_i is in C at time ‘s’
- number[j] was chosen before ‘t2’ and persisted, thus first part of the check could not fail, also ‘i’ and ‘j’ are different, so (number[i],i)<(number[j],j) at time ‘s’

Lemma 2 – mutual exclusion:

Bakery Algorithm satisfies mutual exclusion property

- Assume in contradiction that there are two different processes that have entered critical section
- Then conditions of Lemma 1 are true for both processes symmetrically, that is
  - (number[i],i)<(number[j],j]) and
  - (number[j],j)<(number[i],i):
  - a contradiction
- We conclude that mutual exclusion is satisfied

Lemma 3 – progress

Bakery Algorithm guarantees progress

- Suppose progress is not guaranteed
- Then eventually a point is reached after which all processes are in T or R
- By the code, all the processes in T eventually complete D and reach T-D
- Then the process with the lowest (number,ID) pair is not blocked from reaching C, that is enters critical section
- We conclude that Bakery algorithm satisfies progress

Lemma 4 – fairness

Bakery Algorithm guarantees lockout-freedom

- Consider a particular process P_i in T and suppose it never reaches C
- The process eventually completes D and reaches T-D
- After that any new process that enters D perceives number[i] and chooses a higher number
- Thus, since P_i does not reach C, none of these new processes reach C either
- But by Lemma 3 there must be continuing progress, that is infinitely many entries to C
- Contradiction: P_i blocks the entry to C

Remark on Fairness

- A process that obtained a ticket (number[k],k) will wait at most for (n-1) turns, when other processes will enter the critical section
- For example if all the processes obtained their tickets at the same time they will look like (q,1),(q,2)…(q,n)

  In which case process P_n will wait for processes $P_1 \ldots P_{n\text{-}1}$ to complete the critical section

Bakery alg satisfies "FIFO after a wait-free doorway" fairness property: if Pi completes doorway before Pj enters T, then Pj cannot enter C before Pi. However, it is not FIFO based on time of entry, etc.

RMW shared variables

- Test&set
  f(test&set,v)=(v,1)
- Swap
  f(swap(u),v)=(v,u)
- Fetch& add
  f(fetch&add(u),v)=
  (v,v+u)
- Compare & swap
  f(CAS(u,v),w)
  (w,v) if u=w
  (w,w) otherwise
- LoadLinked & Store Conditional

Load-linked/Store-Conditional

Exchange what is in R4 with (R1)

```
Try:MOV R4, R3   //R3=R4
      LL (R1), R2
      SC R3, (R1)  // if no mem op in betw, SC returns 1 else 0 in R3
      BEQZ R3, Try
      MOV R2, R4
```

Incr a memory value atomically:

```
Try:LL (R1), R2
       INCR R2, R3
       SC R3, (R1)
       BEQZ R3, Try
```

Atomicity

- 2 meanings:
  - no other action in between (typ in OS)
  - either prev state or new state (typ in DB) on failure or visibility of state due to some action
- Interrupts most common reason for lack of atomicity on a uniprocessor
- Some atomicity (& security!) problems
  - Setuid prog (mknod + chown) vs mkdir: on a heavily loaded system: (rm foo; ln /etc/passwd foo) before chown
  - read/write; pread/pwrite; append to a file (lseek + write)
  - open with O_CREAT|O_EXCL : (open + creat)
  - dup2 (fd, fd2) when fd2 open
    (close (fd2) + fcntl(fd, F_DUPFD, fd2))
    - int dup(int fd) returns lowest numbered available fd; -1 on error
    - int dup(int fd, int fd2) returns copy of fd in fd2; closes fd2 if already open; if fd equals fd2, returns fd2 without closing it
  - pselect: signals & select

Old Unix ways of locking (using link)

```
#define LOCKFILE        "seqno.lock"
#include        <sys/errno.h>
extern int      errno;
my_lock() {
        int     tempfd;
        char    tempfile[30];

        sprintf(tempfile, "LCK%d", getpid());

        /* Create a temporary file, then close it. If the temporary file already exists,
           the creat() will just truncate it to 0-length.*/

        if ( (tempfd = creat(tempfile, 0444)) < 0) err_sys("can't creat temp file");
        close(tempfd);

       / * Now try to rename temporary file to the lock file. This will fail if the lock file
           already exists (i.e., if some other process already has a lock). */

        while (link(tempfile, LOCKFILE) < 0) {
                if (errno != EEXIST) err_sys("link error");
                sleep(1);
        }
        if (unlink(tempfile) < 0) err_sys("unlink error for tempfile");
}
my_unlock() {...  unlink(LOCKFILE) ...}
```

Other "Unix" locking

- Note that creat alone cannot be used
  - Creat does not fail if the file EEXISTs (truncs it)
  - Also cannot check if file exists and then creat it
    - Race condition: if((fd=open(file, 0))<0) /*race here*/
      fd=creat(file, 0644) /*rw-r—r-- */
- create the lock file, using open() with both O_CREAT (create file if it doesn't exist) and O_EXCL (error if create and file already exists). If this fails, some other process has the lock.
- Try to create a temporary file, with all write permissions turned off. If the temporary file already exists, the creat() will fail. But: does not work if one of the processes is root!
- Other Probs: crashes do not release locks; how long to wait to retry; no notification to waiters; with busy waiting, priority inversion possible

Concurrency & Locking at Various Levels

- At HW level
  - Instruction level
- At kernel level
  - Between HW events and kernel code
    - Due to Interrupts (interrupt handler and kernel code)
  - Between 2 segments of kernel code
    - Due to true concurrency (SMP)
    - Due to interleaved concurrency (2 procs coroutining or two kernel threads)
- At thread level inside a process
- Across processes on a single machine
- Across multiple machines: at HW/kernel/appl level

Interrupts & Kernel code

- Interrupts (or process scheduling) can occur anytime
- Interrupt handler can also call brelse just like kcode
  - However, interrupt handler should not block
  - Otherwise, the process on whose (kernel) stack the interrupt handler runs blocks
- Can expose data structures in an inconsistent state
  - List manipulation requires multiple steps
  - Interrupt can expose intermediate state
- Interrupt handler can manipulate linked lists that kernel code could also be manipulating
  - Need to raise "processor execution level" to mask interrupts (or scheduling)
  - Check & sleep (or test & set) should be atomic

Sleep and wakeup: Producer-Consumer problem

```
#include "prototypes.h"
#define N 100
int count = 0;
void producer(void) {
  int item;
  while (TRUE)  {
        produce_item(&item);
        if (count == N) sleep();
        enter_item(item);
        count = count + 1;
        if (count == 1)
              wakeup(consumer);
  }
}
```

```
void consumer(void) {
  int item;
  while (TRUE)  {
       if (count == 0) sleep();
       remove_item(&item);
       count = count - 1;
       if (count == N - 1)
        wakeup(producer);
        consume_item(item);
  }
}
```

Race problem if just before sleep, consumer swapped out and producer signals

Semaphores

< await s>0; s -=1 >

- Busy wait solution

```
  down(s) or wait(s): while(s<=0); s -:=1
  up(s) or signal(s): s +=1
```

- Better:

```
 wait(s): s.val -=1;
             if (s.val<0) add (this, s.list); sleep
signal(s): s.val +=1;
             if (s.val<=0) p=remove(s.list); wakeup(p)
```

Semaphores: Solving Producer-Consumer problem

```
semaphore mutex = 1;
semaphore empty = N;
semaphore full  = 0;
void producer(void) {
  int item;
  while (TRUE)  {
        produce_item(&item);
        down(&empty);   ...I
        down(&mutex);   ...II
        enter_item(item);
        up(&mutex);
        up(&full);
  }
}
```

```
void consumer(void){
  int item;
  while (TRUE)  {
        down(&full);
        down(&mutex);
        remove_item(&item);
        up(&mutex);
        up(&empty);
        consume_item(item);
  }
}
```

<u>easily can deadlock if slightly incorrect (eg: exch I & II)</u>

```
initsem(semaphore *sem,  int val) {
   *sem = val
}
void P(semaphore *sem) {
  *sem -= 1;
  while (*sem <0) sleep
}
void V(semaphore *sem) {
  *sem += 1;
  if (*sem<=0) wakeup thread
    blocked on sem
}
boolean_t CP(semaphore *sem) {
  if  (*sem>0) {*sem -= 1; return
    (TRUE) } else return(FALSE)
}
```

- Mutex thru Semaphore

```
semaphore sem;
initsem(&sem, 1);
P(&sem);
use resource
V(&sem);
```

- Event-wait

```
semaphore event;
initsem(&event, 0);
P(&event);
event processing
V(&event);
```

- Countable Resources

```
semaphore counter;
initsem(&counter, count);
P(&counter); use resource; V(&counter)
```

Have we solved the problem?

- P() and V() must be executed atomically
- In uniprocessor system may disable interrupts
- In multi-processor system, use hardware synchronization primitives
  - TS, FAA, etc…
- Involves some limited amount of busy waiting

Event Counters:
Read(E): return curr val of E
Advance(E): atomically incr E by 1
Await(E,v): Wait until E >=v

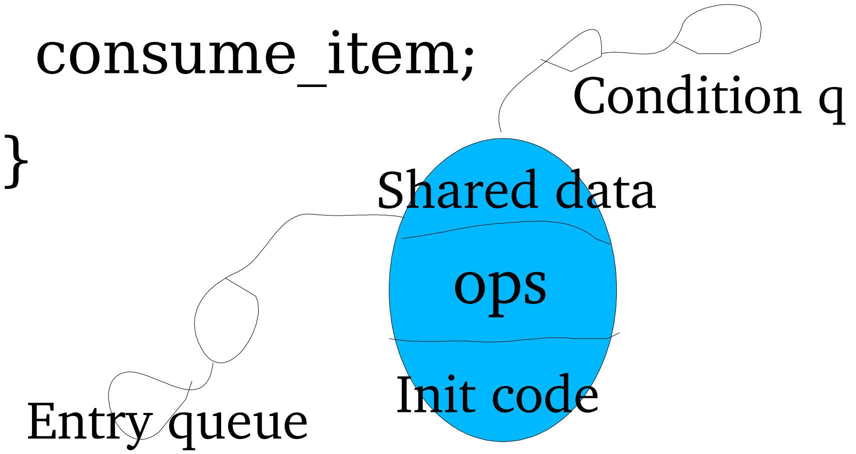
```
event_counter in = 0;
event_counter out = 0;
void producer(void) {
  int item, sequence = 0;
  while (TRUE)  {
        produce_item(&item);
        sequence = sequence + 1;
        await(out, sequence - N);
        enter_item(item);
        advance(&in);
  }
}
```

```
void consumer(void) {
  int item, sequence = 0;
  while (TRUE)  {
        sequence = sequence+1;
        await(in, sequence);
        remove_item(&item);
        advance(&out);
        consume_item(item);
  }
}
```

Monitors

```
monitor ProducerConsumer {
 condition full, empty; integer count;
 void enter() {
   if (count = N) wait(full);
   enter_item;
   count := count + 1;
   if (count = 1)  signal(empty);
 }
 void remove() {
   if (count = 0) wait(empty);
   remove_item;
   count := count - 1;
   if (count = N - 1) signal(full);
 }
 count := 0;
}
```

```
void producer() {
   while (true) {
     produce_item;
     ProducerConsumer.enter,
   }
}
 void consumer() {
   while (true) {
     ProducerConsumer.remove;
     consume_item;
   }
 }
```



Scheduling Issues

- P does x.signal() =>Q may need to be scheduled
  - P waits till the newly scheduled Q leaves monitor or waits for another condition
    - Penalises the signaller!
  - Q waits until P leaves monitor or waits for another condition
    - Used in Java: all locks dropped on wait and regained before returning from wait
      - Has notify and notifyAll
    - Prob: Q's invariant may no longer be true!
  - Concurrent Pascal: compromise
    - P signals only once & exits the monitor; Q starts

Java Monitors

- void wait(); Enter a monitor's wait set until notified by another thread
- void wait(long timeout);   Enter a monitor's wait set until notified by another thread or timeout milliseconds elapses
- void wait(long timeout, int nanos);   Enter a monitor's wait set until notified by another thread or timeout milliseconds plus nanos nanoseconds elapses
- void notify(); Wake up one thread waiting in the monitor's wait set. (If no threads are waiting, do nothing.)
- void notifyAll(); Wake up all threads waiting in the monitor's wait set. (If no threads are waiting, do nothing.)

Java (contd)

- Each Java monitor has a single nameless anonymous condition variable on which a thread can wait() or signal one waiting thread with notify() or signal all waiting threads with notifyAll().
- This nameless condition variable corresponds to a lock on the object that must be obtained whenever a thread calls a synchronized method in the object.
  - Only inside a synchronized method may wait(), notify(), and notifyAll() be called.
- Methods that are static can also be synchronized. There is a lock associated with the class that must be obtained when a static synchronized method is called.

Simulation of a monitor with semaphores

```
typedef int semaphore;
semaphore mutex = 1;
void enter_monitor(void) {
  down(mutex);
}
void leave_normally(void) {
  up(mutex);
}
void leave_with_signal(semaphore c) {
/* signal on c & exit monitor */
  up(c);
}
void wait(semaphore c) {
  up(mutex);
  down(c);
}
```

Message Passing: Mailboxes, Ports, CSP
Send/Receive; Blocking/non-blocking

```
typedef int message[MSIZE];
void producer(void){
  int item;
  message m;
  while (TRUE)  {
        produce_item(&item);
        receive(consumer, &m);
        bulid_message(&m, item);
        send(consumer, &m);
  }
}
```

```
void consumer(void){
  int item, i; message m;
  for (i = 0; i < N; i++)
      send(producer, &m);
  while (TRUE)  {
      receive(producer, &m);
      extract_item(&m, & item);
      send(producer, &m);
      consumer_item(item);
  }
}
```

Problems with Semaphores

- Too complex?
  - Needs low-level atomic op to construct, blocking & unblocking involve context switches, manipulates scheduler and sleep Qs
- Good for resources held for long times, not for short
- Good as V only wakes up if someone can run
- But this can result in convoys
  - Low priority process P1 that has locked an imp lock (L) preempted by P2 which then waits for L
    - Imp lock: Often log lock in txnal systems
    - P3 also needs L, P4 also, ... all wait
  - P1 scheduled again (FIFO) & unlocks L
  - P2 gets lock (P1 preempted), P2 uses lock, then P3, ...
  - For next upd, P2 goes back to Q again, then P3, P4,...
  - Lock-unlock: 100's of insts; lock-wait-dispatch-unlock: 1000's

Active Entities

- User level: multiple processes running at same time (Unix) or one process at any time (DOS) or coop multi-tasking (Win3.1)
- Kernel level: only one ker thread active even with multiple syscalls, exceptions, interrupts: only one running at any time with rest blocked or (Linux 2.x) spinning
  - Or fully pre-emptible kernel (Solaris)
- Processes: "classical" Unix abstraction
  - single addr space, single thread
  - not good enough for TP monitors, db servers
- Threads: user/kernel or LWPs: Solaris & Unixware: m:n model
- SYNCH: not needed for non-preemptive kernels except for blocking: eg: read from file to buffer, buffer needs to be locked for reads (lock, wanted flags)
  - interrupts: raise ipl or block all interrupts
  - MP/RT: need new model

Creating Processes in Unix

- Fork+exec: start new activity executing new code or fork alone: to do diff parts of the same code (servers forking, MP code)
    - id 0 swapper; id1 init; id 2 pagedaemon
    - pid_t get{p,pp}id; {u,g}id_t get{u,eu,g,eg}id; int setuid(uid_t uid); int setr{e,r}uid(uid_t ruid, uid_t euid)
        - real ID: ID of the calling process
        - effective ID: set ID bit on the file being executed
  - May need something simpler (esp in parallel prog)
- Threads
  - Different model: combines fork+exec in one call
  - int pthread_create(pthread_t * thread, pthread_attr_t * attr, void * (*start_routine)(void *), void * arg)

Multiprogramming

- *Multiprogramming*: multiple jobs (processes) in the system
  - Interleaved (time sliced) on a single CPU *OR*
  - Concurrently executed on multiple CPUs
  - Both of the above
- Why multiprogramming?
  - Responsiveness, utilization, concurrency
- Why not?
  - Overhead, complexity
  - Some embedded systems do not have multiple processes or fixed # (may have multiple threads)
- Multi-user?
  - IBM's CMS: no *multi-user*
  - *Win NT (non-server): similar*

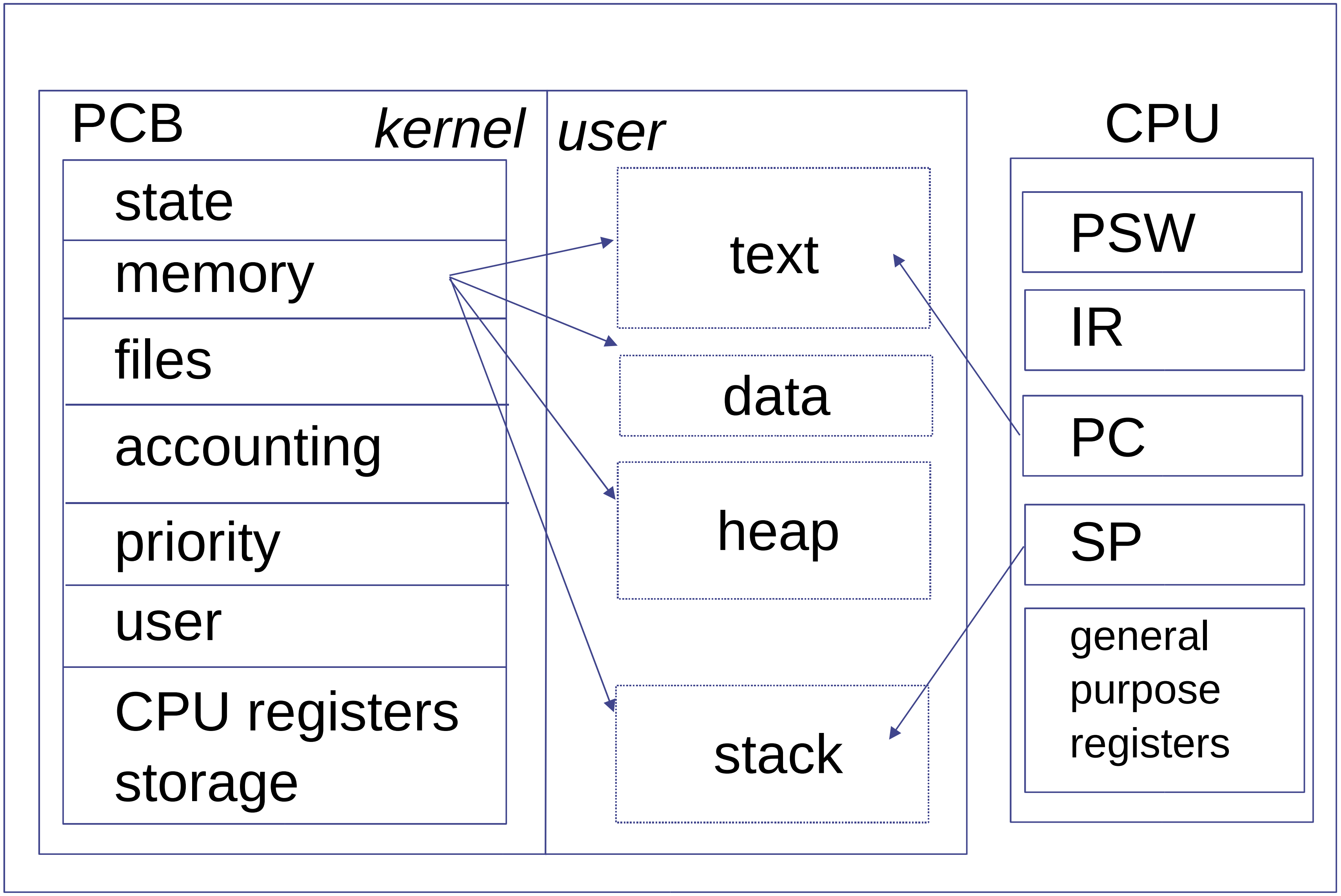
The cost of multiprogramming

- Switching overhead
  - Saving/restoring context wastes CPU cycles
- Degrades performance
  - Resource contention (memory, block buffers,...)
  - Cache/TLB misses
- Complexity
  - Synchronization, concurrency control, deadlock avoidance/prevention

Process Management

- fork, exec (heavy weight), clone (threads)
- Process identity
  - PID, credentials (user ID, group ID), personality (for emulation libraries)
- Process environment
  - Command line arguments; shell variables
- Context
  - Scheduling context (e.g., registers)
  - Accounting info, open file table, file system context, signal handler table, address space info
- Same PCB structure for all process types
  - Thread shares some of data structures of parent
  - Each PCB is just a series of pointers into kernel tables

Process control block (PCB)

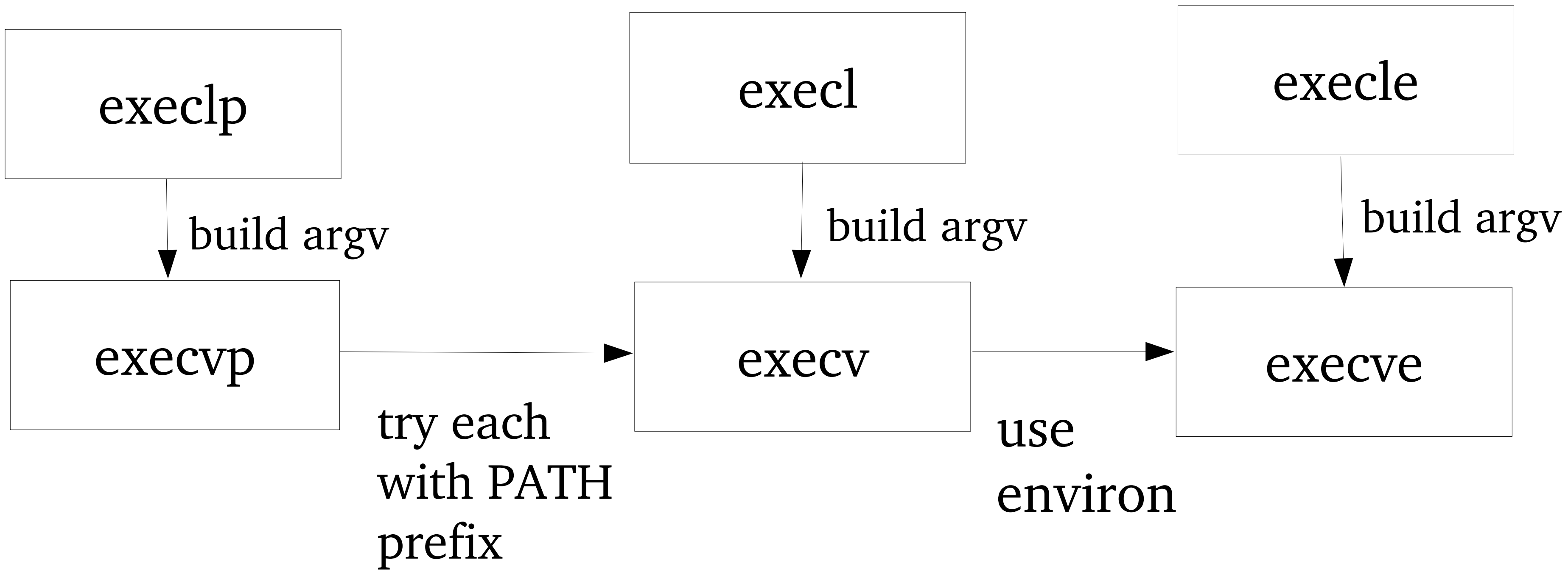


fork

- pid_t fork(void):
  - returns 0 in child and
  - returns pid of child in parent
  - returns -1 on error
- Old impl: copy parent's data,heap, stack; New: copy on write (COW)
  - Child inherits u/gid, eu/gid, seuid/seguid flags; cwd, root dir; umask, signal mask/dispositions; all open fds, close-on-exec flag for all open fds; env, attached shared mem segments, resource limits
  - Before exec, child can redirect stdin/out; close fd's inherited but not needed; change uid/process grp; reset signal handlers
  - Diffs: return val; pid/ppid; file locks; pending alarms/signals cleared; child values for user/sys time
- Vfork: child & parent execute in same addr space
  - Parent blocks until child execs or exits (eg: csh)

Exec

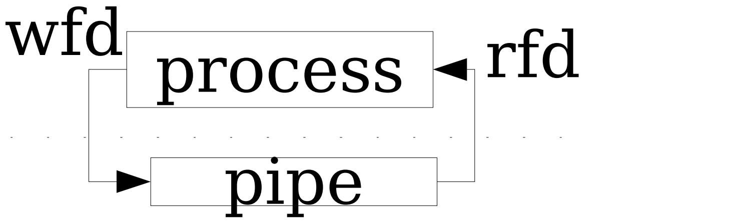
- Execlp + build args-> execvp; execvp+ try with path -> execv
  - P: path has to be looked up (pathify!)
  - int execlp(const char *file, const char *arg, ...)
  - int execvp(const char *file, char *const argv[])
  - int execv(const char *path, char *const argv[])
- Execl + build args-> execv; Execle + build args-> execve
  - L: make a list of args (listify!)
  - int execl(const char *path, const char *arg, ...)
  - int execve(const char *path, char *const argv[], char *const envp[]);
- execv + use environ -> execve
  - E: add explicit env (envify!)
- l2v(v sticks), p2null, add e explicitly!

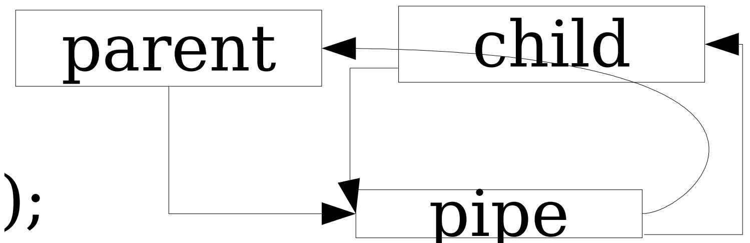

execlp
execl
execle
build argv
build argv
build argv
execvp
execv
execve
try each with PATH prefix
use environ

Death

- Exit (Clib call) or _exit (syscall)
  - Parent dies before child: parent of child=>init
  - Child before: SIGCHLD signal (default disp: IGN)
  - Parent can wait for child (any or specific)
    - Pid_t wait(int * stat_loc)
    - Pid_t waitpid(pid_t pid, int * stat_loc, int options)
    - Till parent waits, a terminated child becomes a zombie
    - When a process inherited by init terminates, init does an implicit wait to fetch termination status: no zombie

```
// int pipe(int fd[2]) returns 2 fds, fd[0] for reading, fd[1] for
// writing with fd[0] connected to fd[1]

int main(void){
        int     n, fd[2];  pid_t   pid;
        char    line[MAXLINE];

        if (pipe(fd) < 0) err_sys("pipe error");
        if ( (pid = fork()) < 0) err_sys("fork error");

        else if (pid > 0) {             /* parent */
                close(fd[0]);
                write(fd[1], "hello world\n", 12);

        } else {                                /* child */
                close(fd[1]);
                n = read(fd[0], line, MAXLINE);
                write(STDOUT_FILENO, line, n);
        }
        exit(0);
}
```





Outline of shell

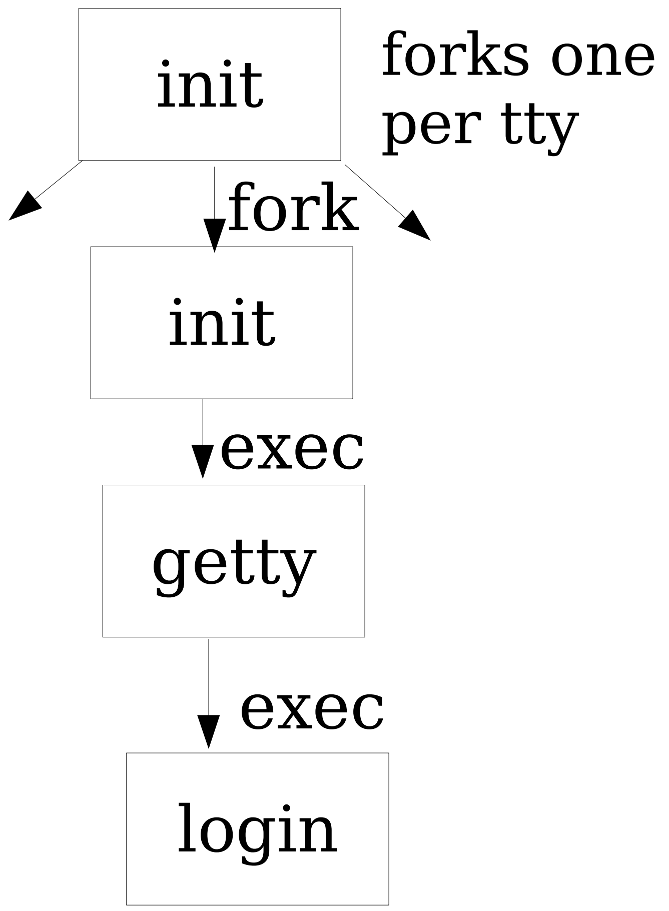
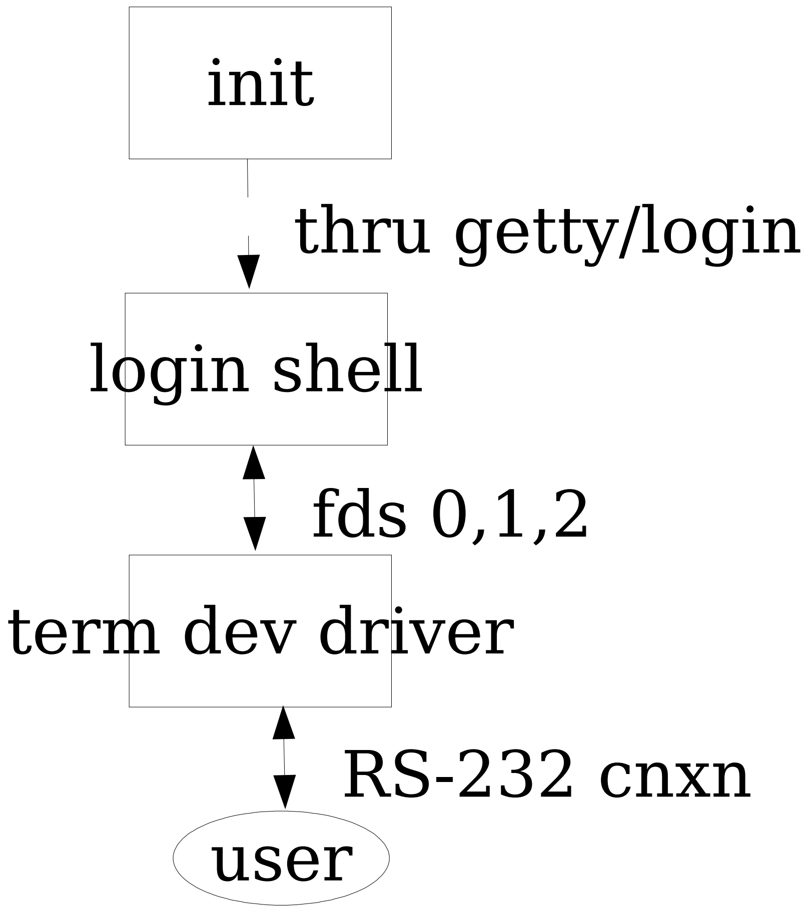
```
while(read(stdin,buffer,nchar)){
  ampr= "& in cmd"
  if cmd cd, etc: execute directly
  if(fork()==0) {
    if (redirect output) {
        fd=creat(newfile, fmask)
        close(stdout); dup(fd); //use
        close(fd)      // dup2(fd,stdout)
    }
    …redirect input and err
    if (pipe) {
      pipe(fd) //creates 2 fds, fd[0]
                // for R and fd[1] W
      if (fork()==0) {
         dup2(fd[1], stdout);
         close(fd[1]);
         close(fd[0]);
          execlp(cmd1, cmd1, 0)
      }
      dup2(fd[0], stdin);
      close(fd[0]);
      close(fd[1]);
    }
    execve(cmd2, cmd2, 0)
  }
  if (!ampr) retid=wait(&status)
}
```

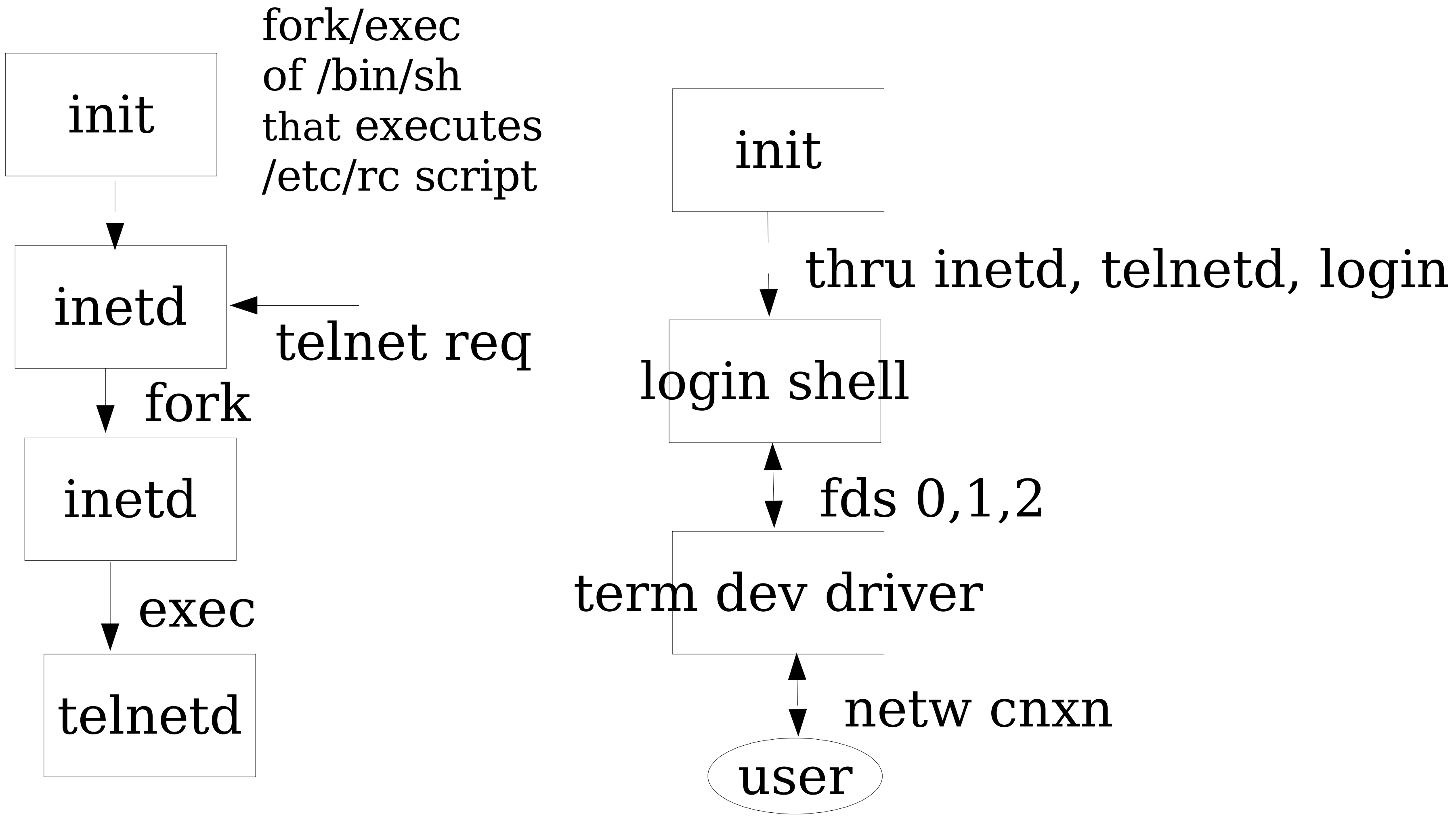
Process Tree

- Init reads /etc/inittab
- Opens tty

```
# Run gettys in standard runlevels
1:2345:respawn:/sbin/mingetty tty1
2:2345:respawn:/sbin/mingetty tty2
3:2345:respawn:/sbin/mingetty tty3
4:2345:respawn:/sbin/mingetty tty4
5:2345:respawn:/sbin/mingetty tty5
6:2345:respawn:/sbin/mingetty tty6
```

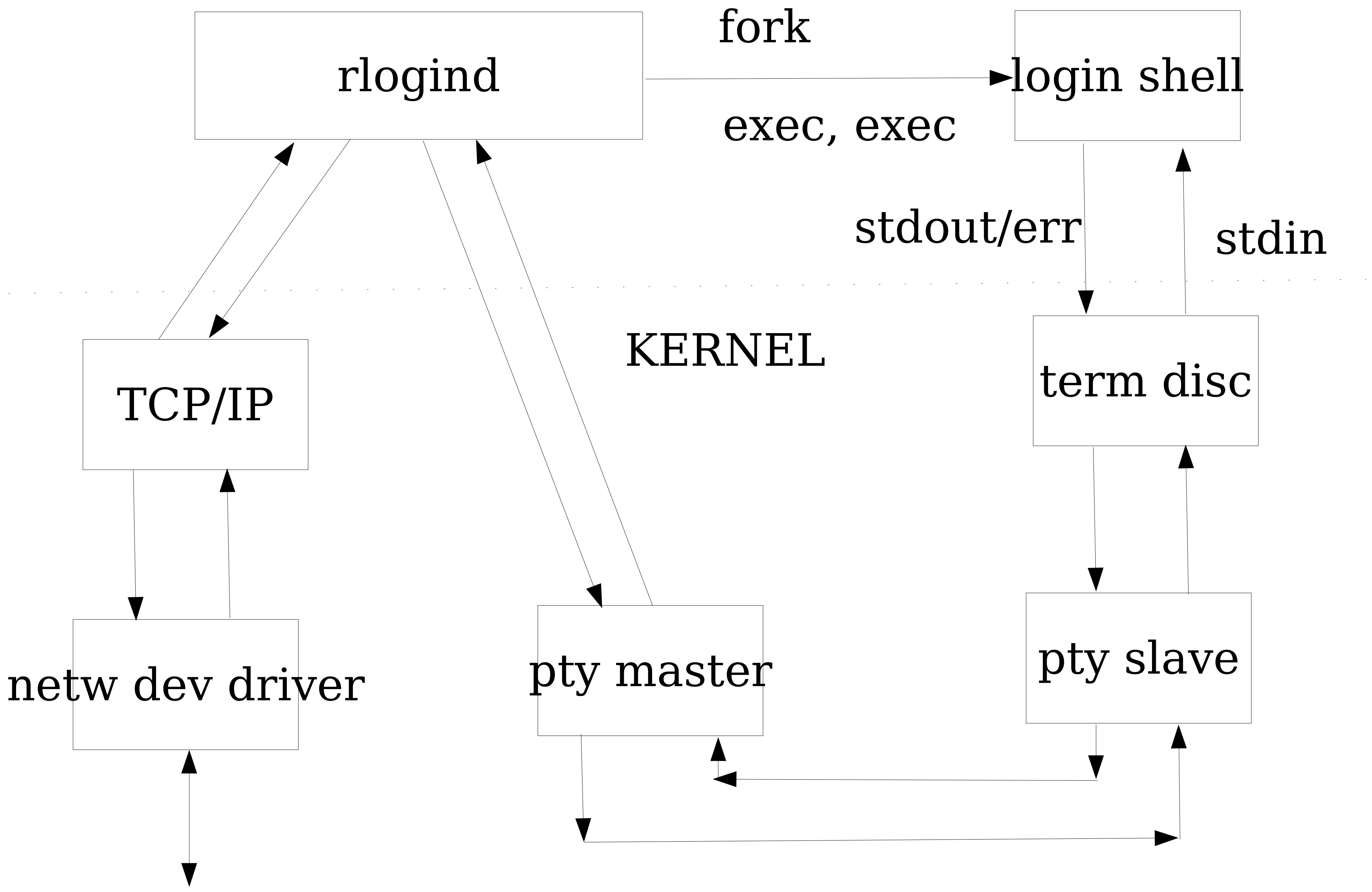
  - Fd 0,1,2 set to dev
  - Login printed
  - Read user name
  - Initial env set (-p: add to existing env; envp: TERM, etc,)
  - uid, gid=0
  - execle("/bin/login", "login", "-p", username, (char*)0, envp)
  - Getpwname (get password file entry); getpass - get a password; use crypt/md5 to validate pwd
  - Fail: login calls exit(1);noticed by init; respawn action
  - Success: chdir; chown for terminal device; setgid; initgroups; initenv (HOME, SHELL, USER, PATH, …)
  - Setuid; then execl("/bin/sh", "-sh", 0) (2nd arg: login shell)


init
forks one per tty
fork
init
exec
getty
exec
login

init
thru getty/login
login shell
fds 0,1,2
term dev driver
RS-232 cnxn
user


init
fork/exec
of /bin/sh
that executes
/etc/rc script
inetd
telnet req
fork
inetd
exec
telnetd
init
thru inetd, telnetd, login
login shell
fds 0,1,2
term dev driver
netw cnxn
user

Network Logins

- Terminal device driver thru, say, RS232
  - Shell (fd 0,1,2): user level
  - Kernel level:
    - Line terminal disc (echo chars, assemble chars to lines, bs, C-u, gen SIGINT/SIGQUIT, C-S, C-Q, newline (CR+LF),...)
    - terminal device driver
- Network login: similar to terminal login
  - init, inetd, telnetd/sshd, login
  - Pseudo-terminal device driver
  - Netw connection thru telnetd/sshd server and telnet/ssh client


rlogind
fork
login shell
exec, exec
stdout/err
stdin
KERNEL
TCP/IP
term disc
netw dev driver
pty master
pty slave