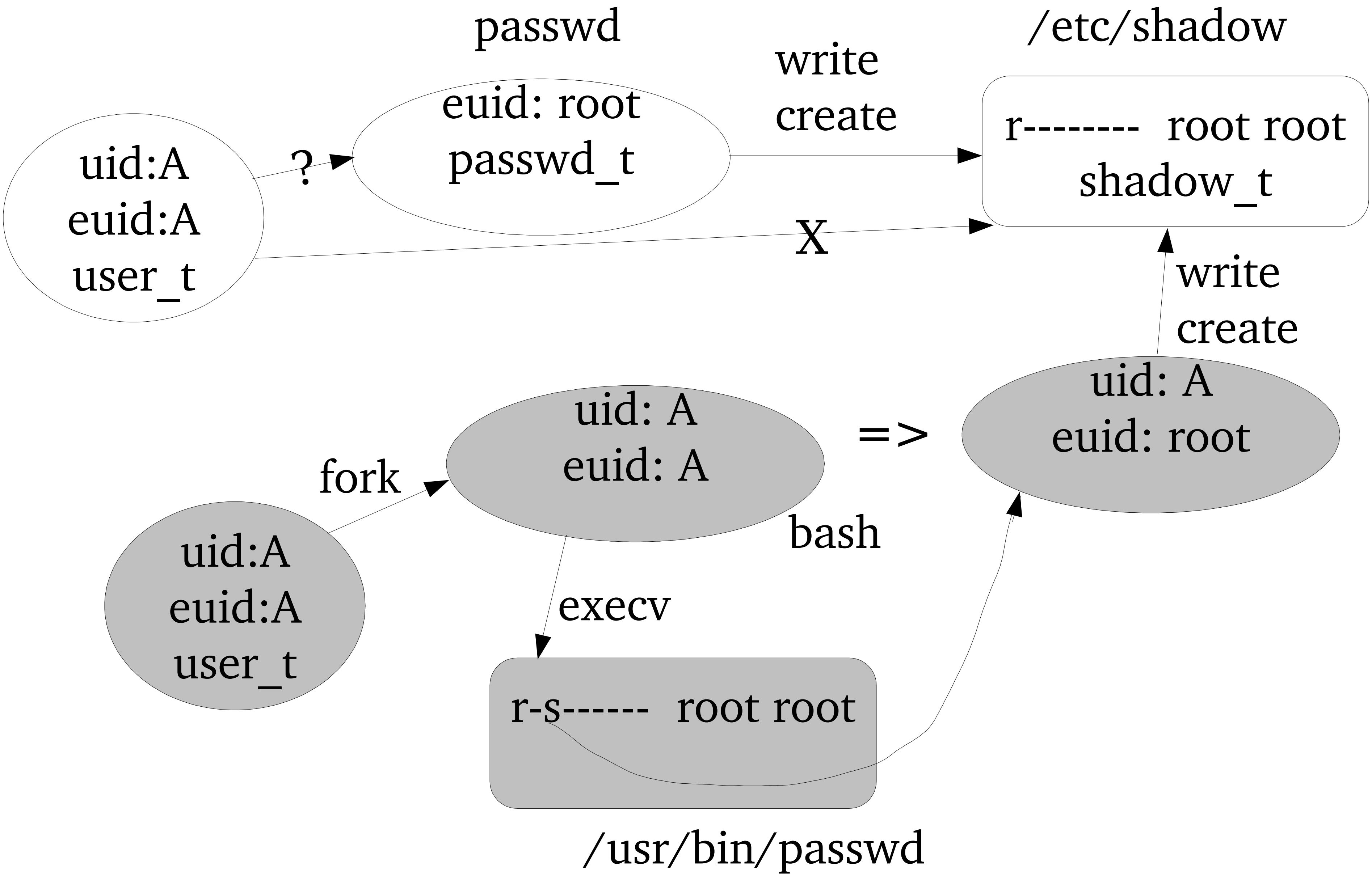
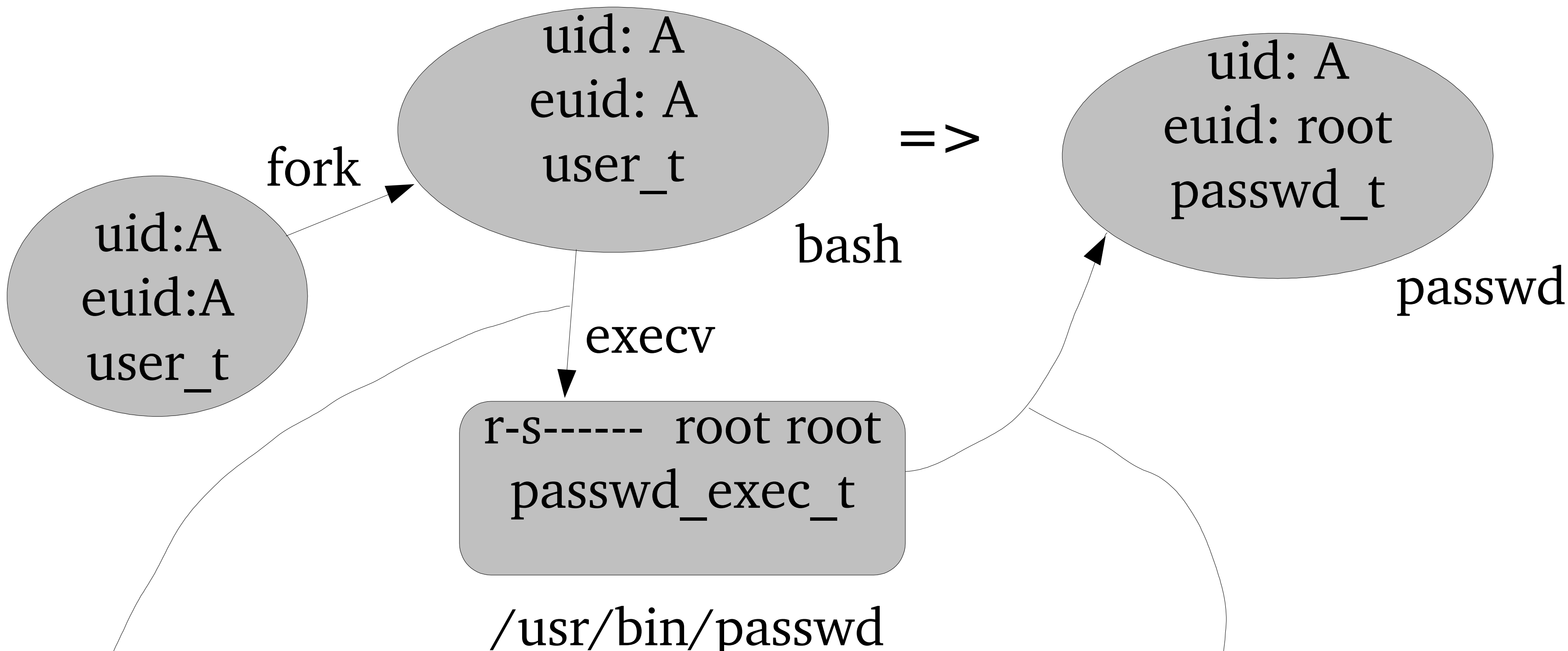
Password mechanism

allow   passwd_t   shadow_t:file   {read, write, append, ...}



domain transitions

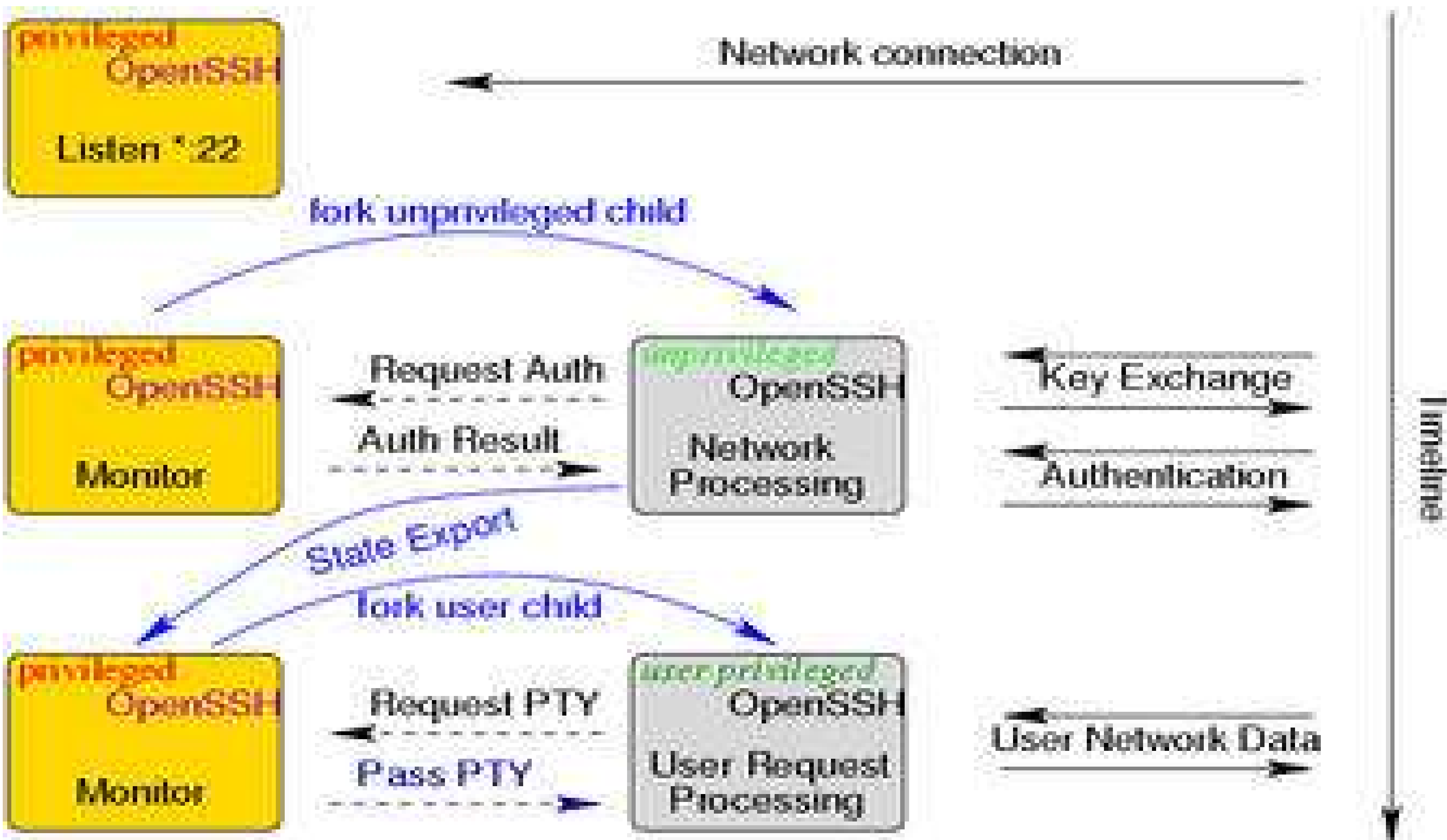


1: allow user_t passwd_exec_t:file {execute, ...}

2: type_transition user_t passwd_exec_t:process passwd_t;
3: allow user_t passwd_t:process transition;
4: allow passwd_t passwd_exec_t:file entrypoint;

Privilege Separation

(from Provos etal, Preventing Privilege Escalation, USENIX Security Symp03)

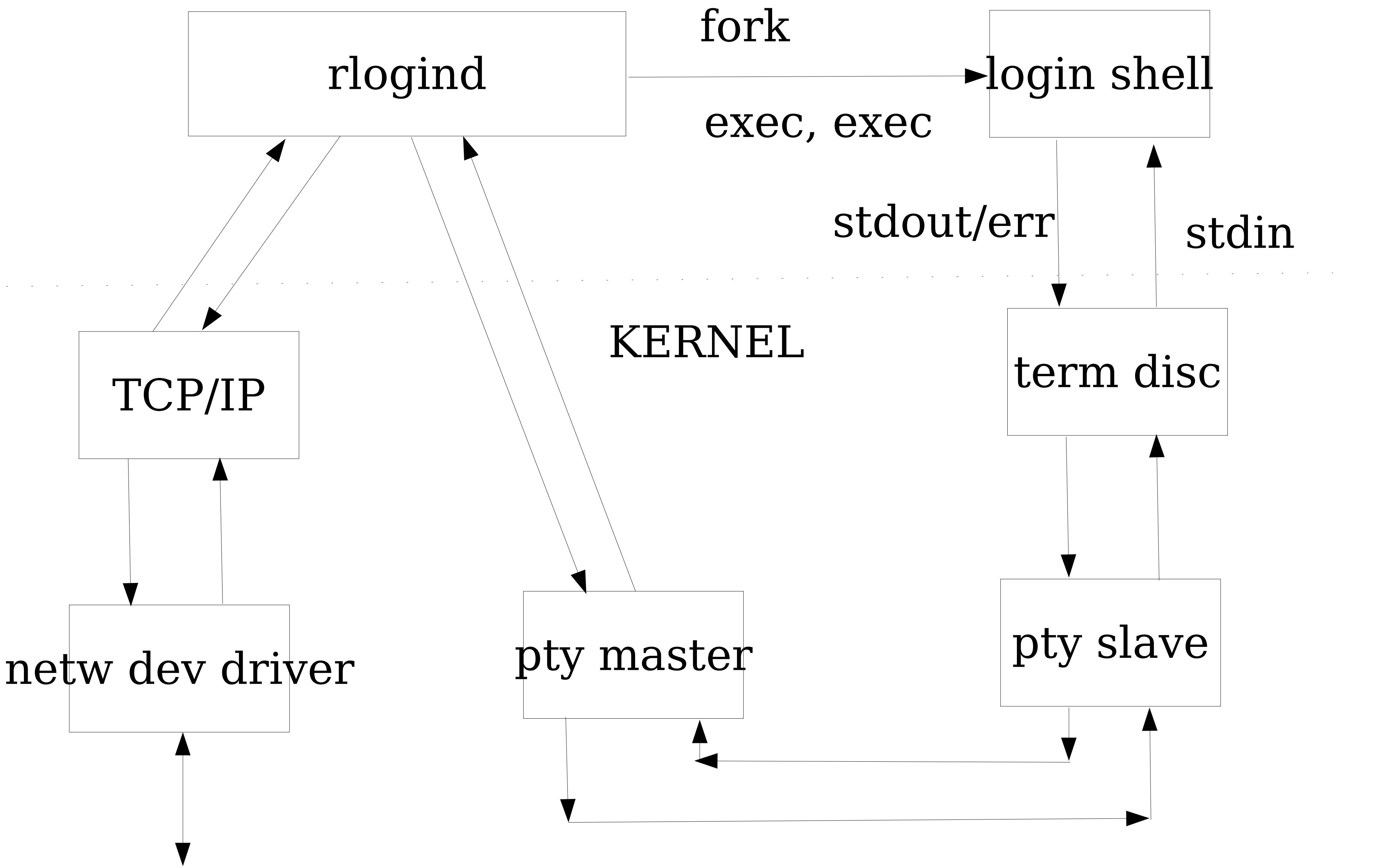


SSH

- Std design
  - On start, SSH daemon binds a socket to port 22 and waits for new cnxns.
  - Every new connection handled by a forked child.
  - Child needs to retain superuser privileges throughout its lifetime to create new pseudo terminals for the user, to authenticate key exchanges when cryptographic keys are replaced with new ones, to clean up pseudo terminals when the SSH session ends, to create a process with the privileges of the authenticated user, etc.
- With privilege separation
  - forked child acts as the monitor
  - monitor forks a slave that drops all its privileges and starts accepting data from the established connection.
  - monitor now waits for requests from the slave. If a request not permitted in the pre-authentication phase issued by the child, monitor terminates.
- Can model the monitor as an FSM

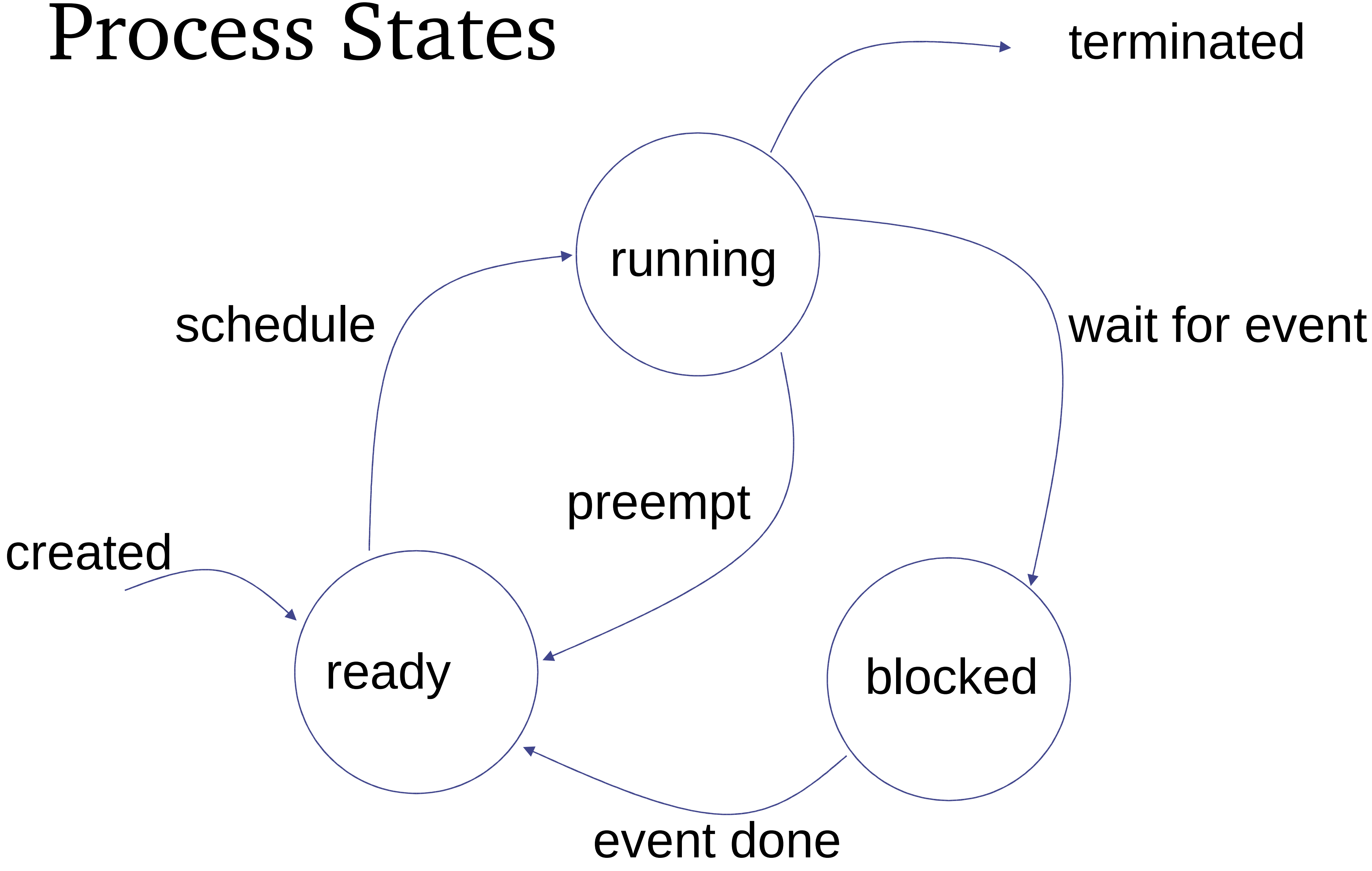
Network Logins

- 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
    - pseudo-terminal is a special IPC that acts like a terminal
    - data written to master side received by the slave side as if it was the result of a user typing at an ordinary terminal & viceversa
  - Netw cnxn thru telnetd/sshd server& telnet/ssh client



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

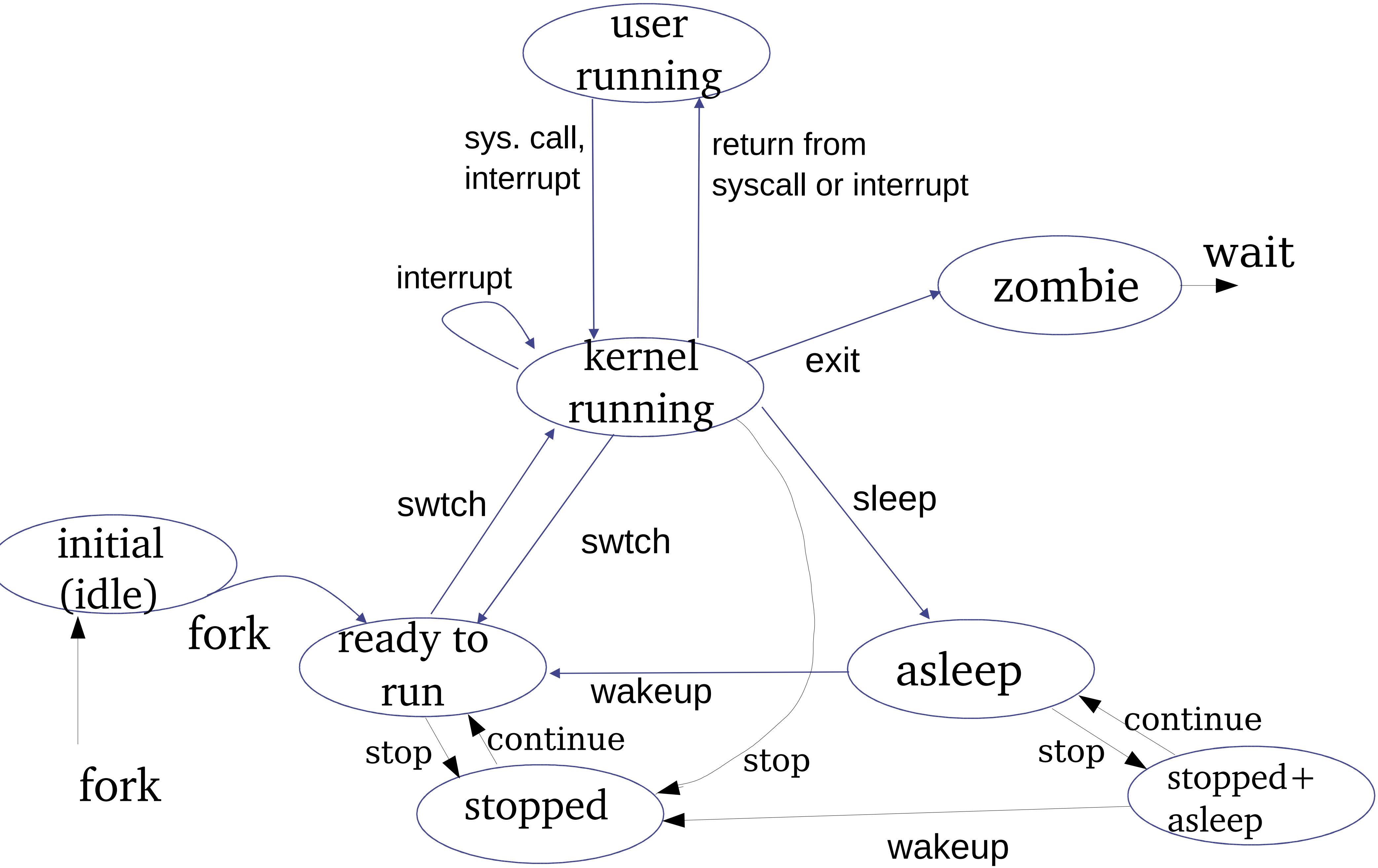
Process States



- asleep:
  - wakeup: ready to run
  - stop: stopped & asleep (4.2+BSD/SVR4)
- stopped:
  - continue: ready to run
- stopped & asleep:
  - continue: asleep
- initial:
  - create (fork): ready to run
- enter ker thru
  - traps/software interrupts (syscall)
  - dev interrupts (disks, terminals, clock),
  - exceptions

- ker running:
  - ret from interrupt, ret from syscall: user running
  - interrupt: still in ker
  - exit: zombie & then wait
  - sleep: asleep
  - swtch: ready to run
  - stop: stopped (4.2BSD+/SVR4) thru SIGSTOP(cannot be caught/blocked/ign)/SIGTSTP(ctrl-Z)/ SIGTTIN/SIGTTOU)
- user running:
  - syscall, interrupt: ker running
- ready to run:
  - swtch: ker running

UNIX Process States



Linux Process states:

- **TASK_RUNNING:** The process is either current or ready to run.
- **TASK_INTERRUPTIBLE:** The process is waiting for an event or resource, but can be woken up by a signal.
- **TASK_UNINTERRUPTIBLE:** The process is waiting directly on hardware conditions and cannot be woken up by signals.
- **TASK_ZOMBIE:** The process has terminated but not removed from task vector yet.
- **TASK_STOPPED:** The process is Stopped.
- **TASK_EXCLUSIVE:** Can be OR-ed with TASK_INTERRUPTABLE or TASK_UNINTERRUPTABLE states: it will be woken up alone instead of all the waiters to avoid the thundering herd problem

Switching Details

- process/context vs mode switch (ker2user & vv)
- process AS: also has
  - u-area (process info of interest to ker: tbl of files opened, savearea, ...): not in Linux
  - proc area:info needed even if process swapped out
  - private ker stack:func call seq can be tracked in ker
- Modes
  - user-mode, process context: applns; sig handlers
  - user-mode, system context: illegal
  - ker-mode, process context : syscalls, exceptions
    kernel can modify AS, u-area, private kernel stack
  - ker-mode, system context : interrupts, system tasks
    ker cannot modify AS, u-area, ker stack of curr process or block

Design Alternatives for Saving State

- Process model: each thread has a ker stack for syscall/exception
  - when thread blocks in ker, stack contains execution state: call sequence+ local vars
  - no need to explicitly save state

```
syscall1 (arg1) {
    ...
    thread_block();
    f2(arg2);
    return;
}
```

- Interrupt Model: Recent 2.6 kernel, V or some RT OS
  - all syscalls+exceptions treated as interrupts
  - single kernel stack per processor for all ker ops
  - to block, must save state (probably in thread/process structure)
    - ker recaptures stack
    - on thread resumption, a new stack allocated and continuation called
  - may be complex: state to be saved may span multiple modules
  - saves stack space: eg: sleep during page fault handling
    - handler code issues a disk read req and blocks
    - after disk read complete, ker retrns thread to user level
    - state to be saved: ptr to page read in, update of mem mapping data

- Can combine both with continuation:
  - thread_block(void (*contfn)())
  - if NULL arg for thread_block, process model. Otherwise, interrupt model.

```
syscall1 (arg1) {
  ...
  save arg1 & other state
  thread_block(f2);
  // not reached
}
f2() {
  restore arg1 & other state
  ...
  thread_syscall_return(status);}
```

Process Subsystem (old!)

- A process is an entry in the process table from the kernel point of view
- Process table: array of task_struct structure accessed as a double-linked list.
  - static array of pointers of length NR_TASKS (a constant defined in include/linux/tasks.h).
  - list structure traversed thru pointers next_task and prev_task
- task_struct contains both low-level and high-level information, ranging from the copy of some hardware registers to the inode of the working directory for the process. (defined in sched.h)
- Current pointer points to task_struct of current running process; it can be modified only by scheduler.

```
extern struct task_struct init_task;
extern struct task_struct *task[NR_TASKS]; //old!
extern struct task_struct *current; //old!
```

task_struct contents:

- Scheduling information: need_resched, counter, nice
- Identifiers (pid, uid, gid, effective uid, effective gid, …)
- Links: orig parent (p_opptr), parent (p_pptr) for ptrace, child (p_cptr), younger/older sib (p_y/osptr), prev_task, next_task
- Times & Timers
- Tty: tty_struct (ttys associated with process)
- File System: fs_struct (cur dir); files_struct (file descriptors for open files)
- Signals: sig_struct
- Virtual Memory: mm_struct
- Process Specific Context (CPU Registers, Stacks, …)
- thread_group: collection of LWPs in a MT application

Stack & Current in Linux 2.4

Stack & process descriptor stored in 2 contig pages (8K)

current points to descr (got by masking 13 lsbits of esp!)

```
static inline struct task_struct * get_current(void) {
 struct task_struct *current;
 __asm__("andl %%esp,%0; ":"=r" (current) : "0" (~8191UL));
 return current;
}
#define current get_current() // process descriptor

union task_union {
        task_t task;
        unsigned long stack[INIT_TASK_SIZE/sizeof(long)];
};

# define INIT_TASK_SIZE 2048*sizeof(long)
```

Threads

- Thread: an execution within a process
- A multithreaded process: many concurrent executions
- Separate: CPU state, stack
- Shared: Everything else: text, data, heap, environment
- Linux: sharing can be fine-grained thru clone
  - int clone(int (*fn)(void *), void *child_stack, int flags, void *arg)   flags: CLONE_PARENT, _FS, _FILES, _SIGHAND, _PTRACE, _VFORK, _VM, _PID, _THREAD
    - Inspired by rfork from Plan9
  - _syscall2(int, clone, int, flags, void *, child_stack);
- int kernel_thread(int (*fn)(void *), void * arg, unsigned long flags){

```
 int p = clone(0, flags| CLONE_VM);
 if (p) return p; /* parent */
 else { fn(arg); exit();}
}
```

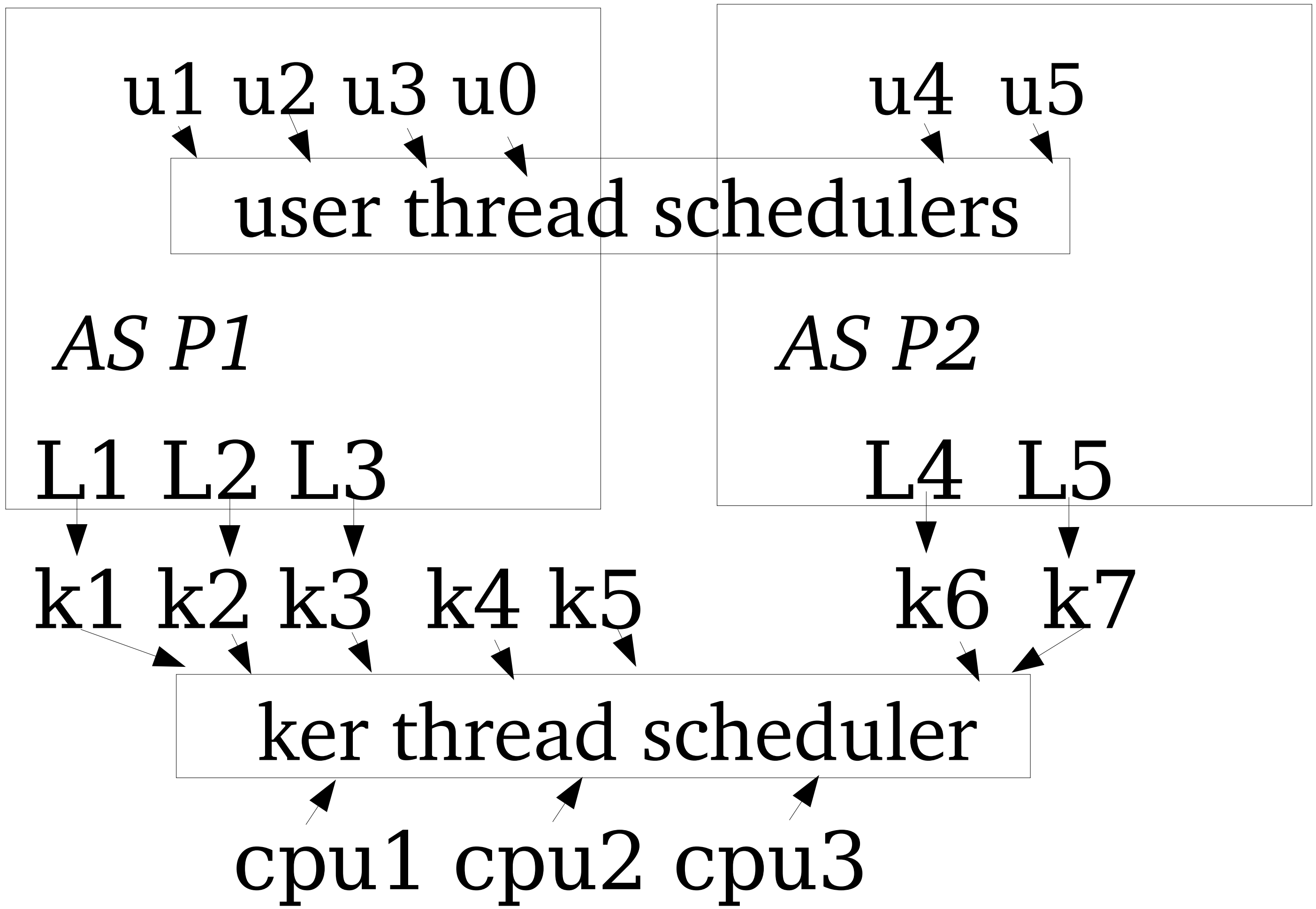
```
int kernel_thread(int (*fn)(void *), void * arg, unsigned long flags) {
        long retval, d0;

        __asm__ __volatile__(
                "movl %%esp,%%esi\n\t"
                "int $0x80\n\t"                 /* Linux/i386 system call */
                "cmpl %%esp,%%esi\n\t"   /* child or parent? */
                "je 1f\n\t"                             /* parent - jump */
                /* Load arg into eax, and push it.  That way, it does
                 * not matter whether called function compiled with
                 * -mregparm or not.  */
                "movl %4,%%eax\n\t"
                "pushl %%eax\n\t"
                "call *%5\n\t"                  /* call fn */
                "movl %3,%0\n\t"        /* exit */
                "int $0x80\n"
                "1:\t"
                :"=&a" (retval), "=&S" (d0)
                :"0" (__NR_clone), "i" (__NR_exit),
                 "r" (arg), "r" (fn),
                 "b" (flags | CLONE_VM)
                : "memory");
        return retval;
```

```
#define _syscall2(type,name,type1,arg1,type2,arg2) \
type name(type1 arg1,type2 arg2) { \
long __res; \
__asm__ volatile ("int $0x80" \
        : "=a" (__res) \
        : "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2))); \
__syscall_return(type,__res); \
}
#define __syscall_return(type, res) \
do { \
        if ((unsigned long)(res) >= (unsigned long)(-125)) { \
                errno = -(res); \
                res = -1; \
        } \
        return (type) (res); \
} while (0)
```

Thread Support

- Operating system
  - Advantage: thread scheduling done by OS
    - Better CPU utilization
  - Disadvantage: overhead if many threads
  - 1-thr, 1-CPU or n-thr, 1-CPU or n-thr, n-CPU
- User-level
  - Advantage: low overhead
  - Disadvantage: not known to OS
    - E.g., a thread blocked on I/O blocks all the other threads within the same process

- ker thread: shares ker text, global data but has own ker stack
  - need not be associated with an user process
  - good for asynch I/O & interrupts
  - need ker stack, save area, sched/synch info
  - context switch fast as mem mappings intact
  - "old" unix: pagedaemon, nfsd
- LWP: ker-supported thread; depends on avlblity of ker threads
  - each LWP indep scheduled, shares AS &other resources with process
  - can make syscalls & block for I/O
  - needs phys mem for ker stack, also reg context, user state & user reg context
  - costly as creating, deleting, synch involves syscalls; significant resources
  - blocking requires ker involvement
  - single LWP impl=> cannot customize for appln



- Linux has only 1-1 threading model
  - Threads are tasks! Need thread group for aggregation

Linux Concurrency Model

- Within appl: clones (incl threads & processes of other systems)
- Inside kernel:
  - Kernel threads: do not have USER context
  - deferrable and interruptible ker funcs:
    - Softirq: reentrant: multiple softirqs of the same type can be run concurrently on several CPUs.
      - No dyn alloc! Have to be statically defined at compile time.
    - Tasklet: multiple tasklets of the same type cannot run concurrently on several CPUs.
      - Dyn alloc OK! Can be allocated and initialized at run time (loadable modules). Impl thru softirqs
    - Bottom Half: multiple bottom halves cannot be run concurrently on several CPUs. No dyn alloc!
      - Impl thru tasklets
- Across HW: IPI

Fork & fork1 in MT processes

- Process with exactly 1 LWP=> same semantics as “old Unix” process
- copy all LWPs on fork? Solaris9 but not Posix
  - one LWP blocked in parent: what about in child? Restart? Concurrent syscalls? EINTR or wait(disk)?
  - one LWP has open netw cnxn: if closed, unexpected user msg to remote node
  - one LWP changing a shared data structure: corruption thru the new copy of LWP? How to make a “consistent” copy?
- copy only calling LWP? Fork1: Solaris10; good for exec'ing
  - some user thrs not on LWPs that were in parent
  - child process should not try to acq locks held by LWPs not in child (deadlock!) but user code cannot know! these locks may be held by ulib POSIX

fork1

fork1(): only calling LWP created in child

- registration of fork_handlers (_atfork)
  - prepare: prior to fork in the ctxt of calling LWP. LIFO
  - parent: after fork. FIFO
  - child: after fork in context of 1 thr in child. FIFO
  - LIFO/FIFO order to enable preserving of locking order

```
int pthread_atfork(void (*prepare)  (void),
void (*parent) (void), void (*child) (void));
```

- handles orphaned mutexes
  - prepare fork handlers lock all mutexes (by calling thr)
  - parent/child fork handlers unlock mutexes
- indep libs & appl progs can protect themselves
  - lib provides fork handlers

Posix Model of Concurrency

- Creation
  - pthread_create(tp, attrp, fptr, argp)
  - pthread_attr_xxx(): manipulate attr of a thread
    - Init/destroy; set/get detachstate, inheritsched, schedparam, schedpolicy, scope, stackaddr, stacksize
- Exit
  - pthread_exit(retvalp)
  - pthread_join(t, **v): wait for another thread termination
  - pthread_detach(t): storage for thread can be reclaimed when thread terminates (no zombie)
- Thread Specific Data (indexed by key)
  - pthread_key_create(keyp, fpdestructor)/_delete()
  - pthread_setspecific()/_getspecific() mapping betw key and thread

- Signal: pthread_sigmask(how, newmask, saveprev): change signal mask for calling thread
  - pthread_kill(t, sig) sigwait: suspend thr till sig
- ID: pthread_self(t)
  - pthread_equal(t1, t2)
  - pthread_once(once?, fptr): ensure some init at most once
- Scheduling
  - pthread_setschedparam()/_getschedparam()
- Cancellation (cancellation pts: _join, _cond_wait, _cond_timedwait, sem_wait, sigwait, _testcancel)
  - pthread_cancel(t) by others /pthread_testcancel(void) by self
  - pthread_setcancelstate()/type()
  - pthread_cleanup_pop()/_push(): if a thread exits or cancelled (with locked mutexes?), cleanup handlers executed; LIFO order

- Mutex
  - pthread_mutex_init()/_destroy()
  - pthread_mutexattr_xxx()
    - Init/destroy; set/get pshared, protocol, prioceiling
  - pthread_mutex_setprioceiling()/_getprioceiling()
  - pthread_mutex_lock()/_trylock()/_unlock()
- Condition Variable
  - pthread_cond_init()/_destroy()
  - pthread_condattr_xxx()
    - Init/destroy; set/get pshared
  - pthread_cond_wait()/_timedwait()
  - pthread_cond_signal()
  - pthread_cond_broadcast()

Condition variables

```
int x,y;
pthread_mutex_t mut =PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
// (waiter) Wait until x is greater than y
        pthread_mutex_lock(&mut);
        while (x <= y) pthread_cond_wait(&cond, &mut);
        /* operate on x and y */
        pthread_mutex_unlock(&mut);
// (signaller) Signal if modifications on x and y st x>y
        pthread_mutex_lock(&mut);
        /* modify x and y */
        if (x > y) pthread_cond_broadcast(&cond);
        pthread_mutex_unlock(&mut);
```

```
// (waiter) if timeout also
struct timeval now;
struct timespec timeout;
int retcode;
pthread_mutex_lock(&mut);
gettimeofday(&now);
timeout.tv_sec = now.tv_sec + 5;
timeout.tv_nsec = now.tv_usec * 1000;
retcode = 0;
while (x <= y && retcode != ETIMEDOUT)
   retcode = pthread_cond_timedwait(&cond, &mut, &timeout);
if (retcode == ETIMEDOUT) {/* timeout occurred */}
else { /* operate on x and y */}
pthread_mutex_unlock(&mut);
```

- Semaphore
  - sem_init()/_destroy()
  - sem_open()/_close()
  - sem_wait()/_trywait()
  - sem_post()
  - sem_getvalue()
  - sem_unlink()
- fork() Clean Up Handling
  - pthread_atfork()
- Async safe? Some pthread calls not safe to call from sig handlers
  - A user thr lib may have taken a lock to ensure, say, that only one user changing Qs. If pthread_mutex_lock, etc, may deadlock

Spinlocks & Semaphores

- Shared data betw different parts of code in kernel
  - most common: access to data structures shared between user process context and interrupt context
- In uniprocessor system: mutual excl by setting and clearing interrupts + flags
- SMP: three types of spinlocks: vanilla (basic), read-write, big-reader
  - Read-write spinlocks when many readers and few writers
    - Eg: access to the list of registered filesystems.
  - Big-reader spinlocks a form of read-write spinlocks optimized for very light read access, with penalty for writes
    - limited number of big-reader spinlocks users.
    - used in networking part of the kernel.
- semaphores: Two types of semaphores: basic and read-write semaphores. Different from IPC's
  - Mutex or counting up()& down(); interruptible/ non

Spinlocks: (cont'd)

- A good example of using spinlocks: accessing a data strucuture shared betw a user context and an interrupt handler

```
spinlock_t my_lock = SPIN_LOCK_UNLOCKED;
my_ioctl() {                        // _ioctl: definitely process context!
     spin_lock_irq(&my_lock);      //  and known that interrupts enabled!
     /* critical section */        //  hence, _irq to disable iinterrupts
     spin_unlock_irq(&my_lock);
}
my_irq_handler() {                  // _irq_handler: definitely system (or intr
     spin_lock(&lock);             // context)& hence known that intr disabled!
     /* critical section */        // can use simpler lock
     spin_unlock(&lock);
}
```

spin_lock: if interrupts disabled or no race with interrupt context

spin_lock_irq: if interrupts enabled and has to be disabled

spin_lock_irqsave: if interrupt state not known

Signals:

- oldest ipc method used by UNIX systems to signal asynchronous events. ONLY 1BIT INFO!
- can be generated by a keyboard interrupt or an error condition or by other processes in the system (if they have the correct privileges)
  - kernel & superuser can send a signal to any process
  - a process can also send a signal to other processes with same uid/gid
- Processes can handle signals themselves or allow kernel to handle
  - If kernel handles the signal, default action for the signal: eg, SIGFPE causes core dump and causes the process to exit
  - SIGSTOP (causes a process to halt its execution) and SIGKILL handled only by kernel
- List of signals on an Linux/Intel machine: SIGHUP SIGINT SIGQUIT SIGILL SIGTRAP SIGIOT SIGBUS SIGFPE SIGKILL SIGUSR1 SIGSEGV SIGUSR2 SIGPIPE SIGALRM SIGTERM SIGCHLD SIGCONT SIGSTOP SIGTSTP SIGTTIN SIGTTOU SIGURG SIGXCPU SIGXFSZ SIGVTALRM SIGPROF SIGWINCH SIGIO SIGPWR

Signals (cont'd)

- void (*signal(int signo, void (*func) (int))) (int) =
- typedef void Sigfunc(int); Sigfunc *signal(int, Sigfunc *)
  - Signal is a func that returns a ptr to a func that ret void (prev sigh)
  - Or, sighandler_t signal(int signum, sighandler_t handler);
- Linux implements signals using information stored in in task_struct of process:
  - struct sigpending pending: currently pending signals
  - blocked: mask of blocked signals
  - struct signal_struct *sig has array of sigactions that holds info about how the process handles each signal
- Signals generated by setting appropriate bit in signal field of pending. If not blocked, scheduler will run handler in the next system scheduling.
- Every time a process exits from a system call, the signal and blocked fields are checked, and if there is any unblocked signal, the handler is called.

```c
#include        <signal.h>
static void  sig_usr(int);   /* one handler for both signals */
int main(void) {
        if (signal(SIGUSR1, sig_usr) == SIG_ERR)
                err_sys("can't catch SIGUSR1");
        if (signal(SIGUSR2, sig_usr) == SIG_ERR)
                err_sys("can't catch SIGUSR2");
        for ( ; ; ) pause(); }
static void sig_usr(int signo) {  /* argument is signal number */
        if (signo == SIGUSR1) printf("received SIGUSR1\n");
        else if (signo == SIGUSR2)
                printf("received SIGUSR2\n");
        else err_dump("received signal %d\n", signo);
        return;}
```

signal: V7, SVR2/3/4 (handler uninstalled, no blocking of signals, no autostart of interrupted system calls)

sigset, sighold, sigrelse, sigignore, sigpause: SVR3/4 (no autostart)

signal, sigvec, sigblock, sigsetmask(unblock a signal), sigpause: 4.x BSD (autostart 4.2; default 4.3/4.4)

sigaction, sigprocmask, sigpending, sigsuspend: autostart unspecified (POSIX.1), optional(SVR4, 4.3/4.4BSD, Linux)

- sigprocmask: change the list of currently blocked signals
- sigpending: allows examination of pending signals (ones which have been raised while blocked)
- sigsuspend: replaces with given signal mask & suspends process until a signal

```
int sigaction(int signo, const struct sigaction *act, struct sigaction *oact)

struct sigaction {
 void (*sa_handler)();
 sigset_t sa_mask; /* addl signals to block */
 int sa_flags; /* restart?, alt stack?, waitchild?, uninstall handler? ...*/ }
```

Unreliable signals

old V7 code: race with a new signal for process before signal reinstalled

```
int sig_int();
...
signal(SIGINT, sig_int);
...

sig_int() {
 /* another signal can come here! can cause default action */
 signal(SIGINT, sig_int);
 ...
}
```

Another race

```
int sig_int_flag;

main() {
 int sig_int();
 ...
 signal(SIGINT, sig_int);
 ...
 while (sig_int_flag==0) /* signal can come here! */ pause();
 ...
}

sig_int() {
 signal(SIGINT, sig_int);
 sig_int_flag=1
}
```

SysV

```
int sighold(int sig); int sigrelse(int sig)

sighold(SIGQUIT); sighold(SIGINT)

c.s.

sigrelse(SIGINT); sigrelse(SIGQUIT)


int sig_int_flag;

main() {

int sig_int();

...

signal(SIGINT, sig_int);

...

sighold(SIGINT);

while (sig_int_flag==0) sigpause(SIGINT);  //atomically release signal

 /* wait for a signal to occur */             // and pause
```

Restarting of interrupted system calls by signals 4.3BSD

Can only call reentrant functions within signal handlers

```
int oldmask;
/* SIGQUIT: quit key + core image; SIGINT: interrupt key ^C */
oldmask= sigblock (sigmask(SIGQUIT)|sigmask(SIGINT));
/* block SIGQUIT/INT */
c.s.
sigsetmask(oldmask) /* reset to old mask */
```

```
int sig_int_flag;
main() { int sig_int();
...
signal(SIGINT, sig_int);
...
sigblock(sigmask(SIGINT));  /* sigblock returns mask before */
while (sig_int_flag==0) sigpause(0); /*wait for signal to occur */
/* sigpause(0) <> sigsetmask + pause as signal can in betw */
/* process signal... */
...
}
```

No Qing for non-real time signals!

```
#include <signal.h>

main() {
    int childPid, i;
    void SigIntHandler();

    sigblock(sigmask(SIGINT));
    signal(SIGINT, SigIntHandler);

    childPid = fork();
    if (childPid > 0) {          /* parent */
        for (i=0; i < 10 ; i++) kill(childPid, SIGINT);
        printf("Parent has issued %d signals to the child\n", i);
    } else {                     /* child */
        sleep(2); /* sleep for 2 secs so that signals overwritten */
        while (1) sigpause(0);
    }
}
void SigIntHandler(int signo) {
    printf("Child : received a signal\n");
}
```

Executing Signal Handlers in Linux

- On signal (either from kernel or another process), ker checks some conditions (disp, etc) before calling do_signal
- do_signal in kernel while (user) signal handler in user mode
- After signal handler run, kernel code executed further
  - However, ker stack no longer contains hw context of interrupted program as ker stack emptied on user mode
  - Also, sig handlers can reenter kernel (syscalls, etc.)
- Solution: copy hw context saved in ker stack to user stack of curr process
  - When sig handler terminates, sigreturn syscall automatically invoked to copy hw context back to kernel stack & restore the user stack
  - Sigframe struct pushed on stack has some code for calling sigreturn: stack has to be executable!!!

pselect

syscall handling

~/csa/os99/udpbcast

Linux Save Structs

- Struct pt_regs //scratch regs; IA-32: all!
  - Minimal state that needs to be saved, say, on dev interrupts
  - IA-64: 2KB but lots of FP; at 2GBps=> 1microsec
    - Ker uses only 4 FP regs => 0.2 microsec
- Struct switch_stack // preserved regs: null! for IA-32
  - Need not be saved (saved by ker funcs if nec)
  - Stack unwinding may be necessary
    - Func may save orig preserved reg on stack but when blocking, saves the curr value in struct
- Struct thread_struct //misc thr state managed lazily
  - debug regs, FP, ... need a fault mech on ker access
  - Ker stack ptr saved in this struct (not lazy!)

```
/* defines the way registers stored on stack during a system call. */

struct pt_regs {
        long ebx;
        long ecx;
        long edx;
        long esi;
        long edi;
        long ebp;
        long eax;
        int  xds;
        int  xes;
        long orig_eax;   // non-neg =>sig has woken up a INTERRUPTIBLE
        long eip;                   //process that was sleeping on a syscall
        int  xcs;
        long eflags;

        long esp;  int  xss; };
```

```
struct thread_struct {
        unsigned long   esp0, eip, esp, fs, gs;
/* Hardware debugging registers */
        unsigned long   debugreg[8];  /* %%db0-7 debug regs */
/* fault info */
        unsigned long   cr2, trap_no, error_code;
/* floating point info */
        union i387_union  i387;
/* virtual 86 mode info */
        struct vm86_struct      * vm86_info;
        unsigned long           screen_bitmap;
        unsigned long           v86flags, v86mask, saved_esp0;
/* IO permissions */
        int ioperm; unsigned long io_bitmap[IO_BITMAP_SIZE+1];
}
```

Internal Thread interface

- void flush_thread(void) <u>*in flush_old_exec*</u>
- start_thread(regs, new_eip, new_esp) <u>in load_elf_binary</u>
- int copy_thread(int nr, unsigned long clone_flags, unsigned long esp, unsigned long unused, struct task_struct * p, struct pt_regs * regs) *in do_fork* to init child's ker stack with CPU regs except eax
- void exit_thread(void) // NULL for x86
- void release_thread(struct task_struct *dead_task) in sys_wait4
- switch_to(prev,next,last) Why last? Else in schedule():
  - A (prev=A, next=B) => B (prev=B, next=C) => C (prev=C, next=A) => A (prev=A, next=**B**)
    - Next is B rather than C! Info about C lost!
    - C's info needed as C may be scheduled to be run elsewhere on a SMP this invocation of schedule
  - When switch_to macro ends, force prev to be C by calling switch_to(prev, next, prev)

```
#define set_fs(x)       (current->addr_limit = (x))
#define start_thread(regs, new_eip, new_esp) do {           \
        __asm__("movl %0,%%fs ; movl %0,%%gs": :"r" (0));   \
        set_fs(USER_DS);                                    \
        regs->xds = __USER_DS;                              \
        regs->xes = __USER_DS;                              \
•       regs->xss = __USER_DS;                              \
        regs->xcs = __USER_CS;                              \
        regs->eip = new_eip;                                \
        regs->esp = new_esp;                                \
} while (0)

//If get_fs() == USER_DS (0x2B), checking is performed; with
//   get_fs() == KERNEL_DS, checking bypassed

void flush_thread(void) {
        struct task_struct *tsk = current;
        memset(tsk->thread.debugreg, 0, sizeof(unsigned long)*8);
        clear_fpu(tsk); /* Forget coprocessor state*/
        tsk->used_math = 0;
}
```

```
#define savesegment(seg,value) asm volatile("movl %%" #seg ",
                                            %0":"=m" (*(int *)&(value)))
retval = copy_thread(0, clone_flags, stack_start, stack_size, p, regs);
        // first 2 args historical; not needed now!
int copy_thread(int nr, unsigned long clone_flags, unsigned long esp,
    unsigned long unused,  struct task_struct * p, struct pt_regs * regs){
        struct pt_regs * childregs;
        childregs =((struct pt_regs *)(THREAD_SIZE +(unsigned long)p)) -1;
        struct_cpy(childregs, regs);
        childregs->eax = 0; // return val
        childregs->esp = esp;

        p->thread.esp = (unsigned long) childregs;
        p->thread.esp0 = (unsigned long) (childregs+1);

        p->thread.eip = (unsigned long) ret_from_fork;

        savesegment(fs,p->thread.fs);
        savesegment(gs,p->thread.gs);

        unlazy_fpu(current);
        struct_cpy(&p->thread.i387, &current->thread.i387);
        return(0);}
```

Context Switch

- Happens thru calls to schedule(). 550 of them! In cpu_idle, __down, sys_sigsuspend, do_signal, sys_pause but many in driver code

```
void cpu_idle (void) { /* endless idle loop with no priority at all */
        while (1) {
                void (*idle)(void) =pm_idle;//power mgmt idle func
                if (!idle) idle = default_idle;
                if (!current->need_resched) idle();
                schedule();
                check_pgt_cache() //frees pages if excess in cache;}}
asmlinkage int sys_pause(void) {
        current->state = TASK_INTERRUPTIBLE;
        schedule();
        return -ERESTARTNOHAND;}
```

```
...     regs->eax = -EINTR;         // from sys_sigsuspend
        while (1) {
                current->state = TASK_INTERRUPTIBLE;
                schedule();
                if (do_signal(regs, &saveset)) return -EINTR;
        }
...no_signal:  /* Did we come from a syscall? */ // in do_signal
        if (regs->orig_eax >= 0) {
                /* Restart the system call - no handlers present */
                if (regs->eax == -ERESTARTNOHAND ||
                    regs->eax == -ERESTARTSYS ||
                    regs->eax == -ERESTARTNOINTR) {
                        regs->eax = regs->orig_eax;
                        regs->eip -= 2;}}
```

need_resched

- need_resched set on do_fork to let child run first to avoid most of the COW overhead when the child exec()s afterwards
- Also if task's quantum might have expired already, but not scheduled off yet; thru resched_task, poll_idle, etc.

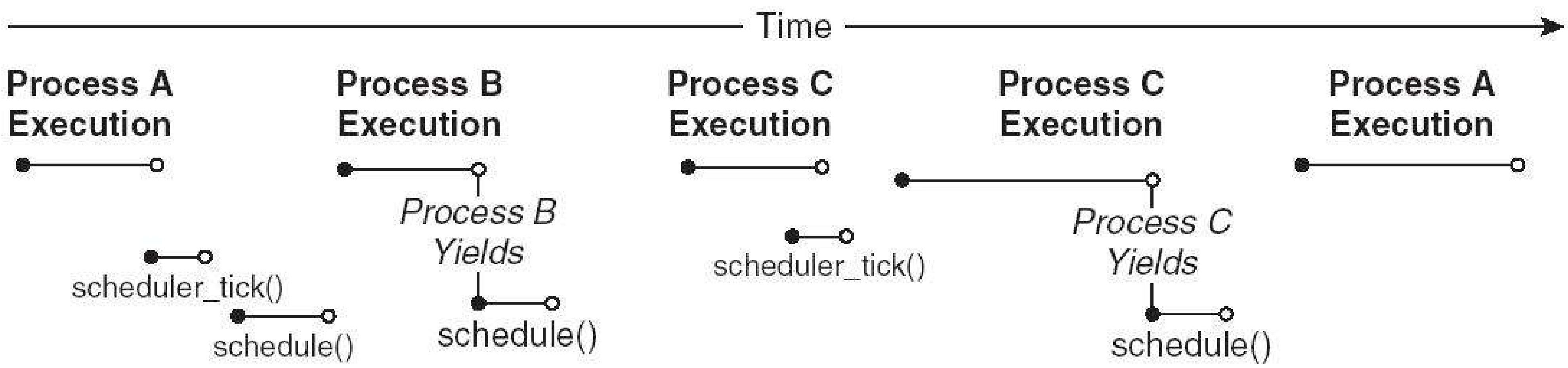
```
static void poll_idle (void){   int oldval;
        __sti();
         /* another CPU has just chosen a thread to run here? */
        oldval = xchg(&current->need_resched, -1);
        if (!oldval)
                asm volatile(
                        "2:"
                        "cmpl $-1, %0;"
                        "rep; nop;"   SIX times!
                        "je 2b;"    : :"m" (current->need_resched));}
```

```
void default_idle(void){
        if (current_cpu_data.hlt_works_ok && !hlt_counter) {
                __cli();
                if (!current->need_resched) safe_halt();
                else __sti();
        }
}
#define __sti()         __asm__ __volatile__("sti": : :"memory")
#define __cli()         __asm__ __volatile__("cli": : :"memory")
#define safe_halt()     __asm__ __volatile__("sti; hlt": : :"memory")
struct prio_array {
        int nr_active;
        unsigned long bitmap[BITMAP_SIZE];
        list_t queue[MAX_PRIO];
}; //prio_array_t
```

```
#define switch_to(prev,next,last) do {   //last is in ebx                              \
        asm volatile("pushl %%esi\n\t"   \       //esi, edi,ebp preserved
                     "pushl %%edi\n\t"         \
                     "pushl %%ebp\n\t"         \
                     "movl %%esp,%0\n\t" /* save ESP  to prev->thread.esp */  \
                     "movl %2,%%esp\n\t" /*restore ESP f. next->thread.esp: new stack!*/\
                     "movl $1f,%1\n\t"      /* save EIP into  prev->thread.eip*/   \
                     "pushl %3\n\t"          /* push next->thread.eip */ \
                     "jmp __switch_to\n"                               \
                     "1:\t"  "popl %%ebp\n\t"                                  \
                     "popl %%edi\n\t"                                  \
                     "popl %%esi\n\t"                                  \
                     :"=m" (prev->thread.esp)(%0),"=m" (prev->thread.eip)(%1)//upd mem\
                     :"m" (next->thread.esp)(%2),"m" (next->thread.eip)(%3), //read mem  \
                      "a" (prev)(%eax), "d" (next)(%edx));  // saves eax/edx implicitly!        \
} while (0)  // note: bold chars are embedded comments!!!
```

__switch_to()

- updates
  - The next thread structure with kernel stack pointer
  - Thread local storage descriptor for this processor
  - fs and gs for prev and next, if needed
  - Debug registers, if needed
  - I/O bitmaps, if needed
- __switch_to() then returns upd prev task struct



```
asmlinkage void schedule(void) {
        task_t *prev, *next;
        runqueue_t *rq;
        prio_array_t *array;
        list_t *queue;
        int idx;
        if (unlikely(in_interrupt()))
         BUG();
need_resched:
        prev = current;
        rq = this_rq();
        release_kernel_lock(prev,
           smp_processor_id());
        prepare_arch_schedule(prev);
        prev->sleep_timestamp = jiffies;
        spin_lock_irq(&rq->lock);
```

```
switch (prev->state) {
case TASK_INTERRUPTIBLE:
  if (unlikely(signal_pending(prev))){
      prev->state = TASK_RUNNING;
      break;}
default: deactivate_task(prev, rq);
case TASK_RUNNING: ;
}
#if CONFIG_SMP
pick_next_task:
#endif
    if (unlikely(!rq->nr_running)) {
#if CONFIG_SMP
    load_balance(rq, 1);
    if (rq->nr_running)
         goto  pick_next_task;
#endif
```

```
    next = rq->idle;
   rq->expired_timestamp = 0;
   goto switch_tasks;
 }
 array = rq->active;
 if (unlikely(!array->nr_active)) {
/* Switch active & expired arrays.*/
     rq->active = rq->expired;
     rq->expired = array;
     array = rq->active;
     rq->expired_timestamp = 0;
 }
  idx = sched_find_first_bit
                   (array->bitmap);
  queue = array->queue + idx;
  next = list_entry(queue->next,
   task_t, run_list);
```

```
switch_tasks:
   prefetch(next);
   clear_tsk_need_resched(prev);
   if (likely(prev != next)) {
      rq->nr_switches++;
      rq->curr = next;
      prepare_arch_switch(rq);
      prev=context_switch(prev, next)
      barrier();
      rq = this_rq();
      finish_arch_switch(rq);
    } else spin_unlock_irq(&rq->lock);
   finish_arch_schedule(prev);
   reacquire_kernel_lock(current);
   if (need_resched())
        goto need_resched;}
```

```c
static inline void deactivate_task(struct task_struct *p, runqueue_t *rq) {
        rq->nr_running--;
        if (p->state == TASK_UNINTERRUPTIBLE) rq->nr_uninterruptible++;
        dequeue_task(p, p->array);
        p->array = NULL;
}
static inline void dequeue_task(struct task_struct *p, prio_array_t *array) {
        array->nr_active--;
        list_del(&p->run_list);
        if (list_empty(array->queue + p->prio))
                __clear_bit(p->prio, array->bitmap); // no tasks at prio priority
}
static inline int need_resched(void){
        return unlikely(current->need_resched);
}
#define this_rq()    cpu_rq(smp_processor_id())
```

```
static inline task_t * context_switch(task_t *prev, task_t *next) {
        struct mm_struct *mm = next->mm;
        struct mm_struct *oldmm = prev->active_mm;
        if (unlikely(!mm)) {  // kernel thread
                next->active_mm = oldmm; // borrow old AS to avoid flushing tlb
                atomic_inc(&oldmm->mm_count);
                enter_lazy_tlb(oldmm, next, smp_processor_id()); //
        } else
                switch_mm(oldmm, mm, next, smp_processor_id());
        if (unlikely(!prev->mm)) { //kernel thread
                prev->active_mm = NULL; // unborrow borrowed AS!
                mmdrop(oldmm);
        }
        /* Here we just switch the register state and the stack. */
        switch_to(prev, next, prev);
        return prev; }
```

```
#define in_interrupt() ({ int __cpu = smp_processor_id(); \
        (local_irq_count(__cpu) + local_bh_count(__cpu) != 0); })


#ifdef CONFIG_SMP
#define __IRQ_STAT(cpu, member) (irq_stat[cpu].member)
#else
#define __IRQ_STAT(cpu, member) ((void)(cpu), irq_stat[0].member)
#endif
  /* arch independent irq_stat fields */
#define softirq_pending(cpu)    __IRQ_STAT((cpu), __softirq_pending)
#define local_irq_count(cpu)    __IRQ_STAT((cpu), __local_irq_count)
#define local_bh_count(cpu)     __IRQ_STAT((cpu), __local_bh_count)
#define syscall_count(cpu)      __IRQ_STAT((cpu), __syscall_count)
#define ksoftirqd_task(cpu)     __IRQ_STAT((cpu), __ksoftirqd_task)
```

Verifying the User Parameters

- All syscall params must be checked before user's req satisfied by kernel
  - Mem checks common to almost all syscalls
- Verify that linear addr in UAS with corr perms
  - < PAGE_OFFSET (above ker AS) a)
  - Inside UAS, in mapped region b)
  - Time consuming (even corr syscalls penalized!)
- Linux 2.2+: only a) done thru verify_area

```
#define access_ok(addr,size) ({ unsigned long flag, sum; \
    asm("addl %3,%1 ; sbbl %0,%0; cmpl %1,%4; sbbl $0,%0" \
    :"=&r" (flag), "=r" (sum):"1" (addr),"g" ((int)(size)),"g" (current->addr_limit.seg));
      flag; })    //code eq to: addr+size<addr || addr+size >current->addr_limit.seg
static inline int verify_area(int type, const void * addr, unsigned long size) {
   return access_ok(type,addr,size) ? 0 : -EFAULT;}
```

```
#define get_user(x,ptr)   // no checking!                    \
({      int __ret_gu,__val_gu;                               \
        switch(sizeof (*(ptr))) {                            \
        case 1:  __get_user_x(1,__ret_gu,__val_gu,ptr); break;     \
        case 2:  __get_user_x(2,__ret_gu,__val_gu,ptr); break;     \
        case 4:  __get_user_x(4,__ret_gu,__val_gu,ptr); break;     \
        default: __get_user_x(X,__ret_gu,__val_gu,ptr); break;     \
        }                                                    \
        (x) = (__typeof__(*(ptr)))__val_gu;          \
        __ret_gu;                                            \
})
#define __get_user_x(size,ret,x,ptr) \
        __asm__ __volatile__("call __get_user_" #size \
                :"=a" (ret),"=d" (x) :"0" (ptr))
```

```
#define __get_user_size(x,ptr,size,retval)  do { retval = 0;
        switch (size) {                                              \
          case 1: __get_user_asm(x,ptr,retval,"b","b","=q"); break;     \
          case 2: __get_user_asm(x,ptr,retval,"w","w","=r"); break;     \
          case 4: __get_user_asm(x,ptr,retval,"l","","=r"); break;      \
          default: (x) = __get_user_bad();}} while (0)
#define __get_user_asm(x, addr, err, itype, rtype, ltype)  __asm__ __volatile__(
                "1:     mov"itype" %2,%"rtype"1\n"              \
                "2:\n"                                          \
                ".section .fixup,\"ax\"\n"                      \
                "3:     movl %3,%0\n"                           \
                "       xor"itype" %"rtype"1,%"rtype"1\n"       \
                "       jmp 2b\n"                               \
                ".previous\n"                                   \
                ".section __ex_table,\"a\"\n"                   \
                "       .align 4\n"                             \
                "       .long 1b,3b\n"                          \
                ".previous"                 : "=r"(err), ltype (x) : "m"(__m(addr)), "i"(-EFAULT), "0"(err))
```