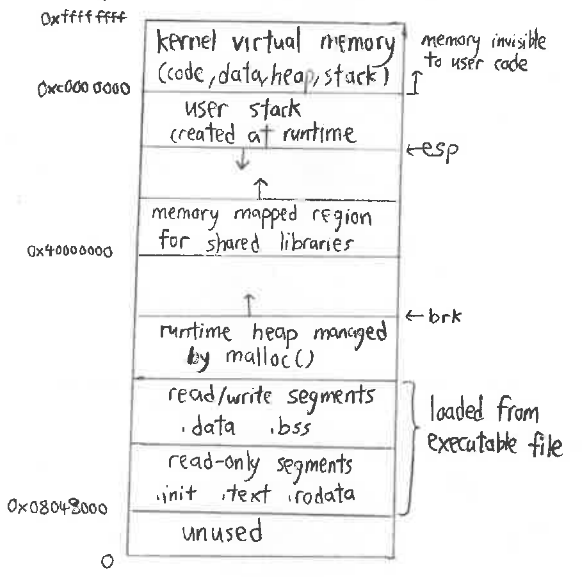
L2/Stack

OS will be 32-bit, not 64-bit

- simpler for kernel, machine boot goes from 16 → 32 → 64 bit
- simpler interrupt handling
- simpler debugging (less registers)
- simpler virtual memory

Each process has its own private address space

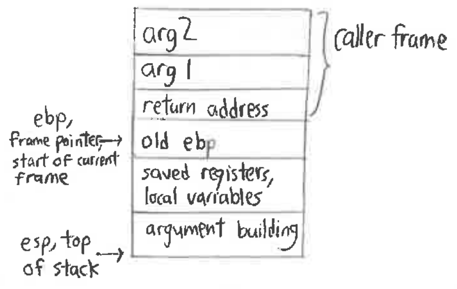

0xffffffff
kernel virtual memory (code, data, heap, stack)
memory invisible to user code
0xc0000000
user stack created at runtime
←esp
memory mapped region for shared libraries
0x40000000
←brk
runtime heap managed by malloc()
read/write segments .data .bss
loaded from executable file
read-only segments .init .text .rodata
0x08048000
unused
0

Private address space layout for IA32 Linux 2.x.
F KVM C0 USER STACKTOP 40 MMAPSHAREDLIBBOT
08048000 PROGRAMBOT

Stack Manipulation

- Stack = region of memory managed with stack discipline
- pushl src : fetch operand from src, dec esp by 4, store
- popl dst : read memory at esp, inc esp by 4, load
- call label : push return address, jump to label
- ret : pop address from stack, jump to address
- To support recursion, code must be reentrant.
- State only needed for limited time, callee returns before caller does.
- Hence stack is allocated in nested frames.

Stack Frame.



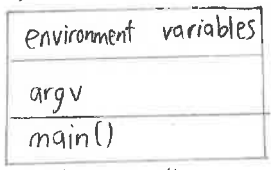
Register Saving Conventions

- Caller save : caller saves temporary in frame before calling
- Callee save : callee saves temporary in frame before using

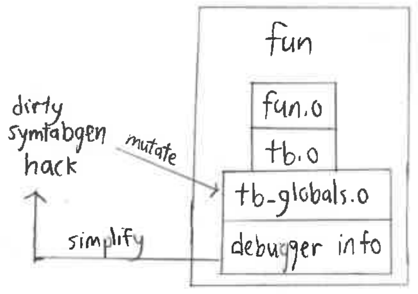
caller save { eax ← also used for return value, edx, ecx }
callee save { ebx, esi, edi }
special { esp, ebp }
} User registers, note that non-user registers exist but are not mentioned here

Running Programs

- OS copies argc, argv from old address space to new address space in exec(), typically below bottom of stack with other weird things
- main()'s return(0); defined to have the same effect as exit(0);
- Accomplished by wrapping main() in "crt0.s", void ~~main(int argc, char *argv[]) { exit(main(argc, argv)); }

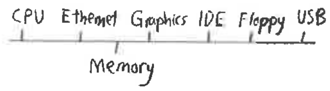


P0

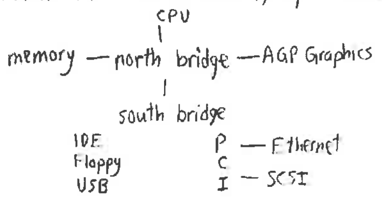


L3b / Hardware

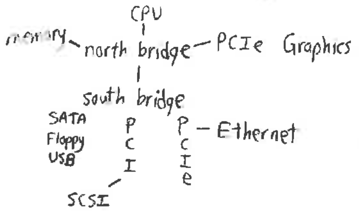
Historically, one shared bus.



Then in 1997-2004, split bus.



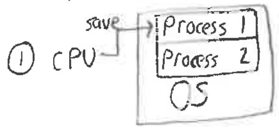
Then in 2004-2008, further split.



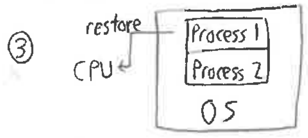
Kernel Mode

- Programming language runtimes differ
  ML => only heap, C => stack based
- The processor is agnostic, though some processors mandate a stack
- Trap handler (INT or software interrupt in Intel parlance) responsible for:
  1. Switch to correct stack
  2. Save registers
  3. Enable virtual memory
  4. Flush caches

Example. getpid() by Process 1

1. 

2. running→u_reg[R_EAX] = running→u_pid;
executed in kernel mode

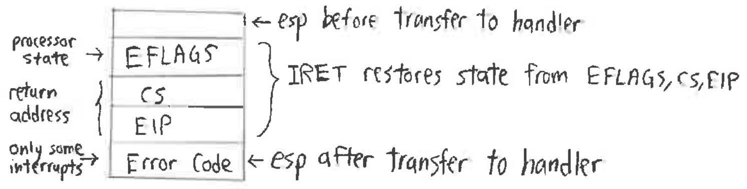
3. 

Opportunity for multitasking!

Process 1 read()
↓
Kernel tells disk to read sector xyz
↓
Kernel switches to running P2, marks P1 as blocked
→ Disk done reading, issues interrupt request. CPU stops running P2, interrupts to kernel, run disk interrupt handler
↑
Kernel switches back to process 1, P2 marked as runnable

Handling Interrupts

- Interrupt vector table
  - Table base pointer programmed in OS startup
  - Table entry size defined by hardware
  - interrupt controller's interrupt source number ⇒ interrupt table entry for interrupt dispatch
    1. Save old processor state
    2. Modify CPU state per table entry, crucially program counter + status register which define the new execution environment
    3. Start running interrupt handler
    4. On interrupt return, load saved processor state back into registers
    5. Restore program counter to reactivate old code
    6. Hardware instruction normally restores some state, kernel must restore remainder
- State is normally stored on the kernel stack. The interrupt handler uses the kernel stack as scratch space, and interrupt return IRET will load registers from the kernel stack too. IRET may also switch mode from kernel to user.


← esp before transfer to handler
processor state →
EFLAGS
return address
CS
EIP
IRET restores state from EFLAGS, CS, EIP
only some interrupts →
Error Code
← esp after transfer to handler

Example stack usage for an interrupt while in kernel mode, no privilege change.

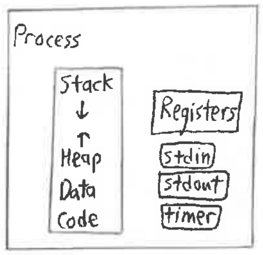
- Interrupt handler can cause race conditions if it modifies state that the user process uses as a guard

if (device_idle) {A} else {B}

User	Interrupt handler
if (device_idle) /* no, go to B */	INTERRUPT, set device_idle = 1
B - error!	

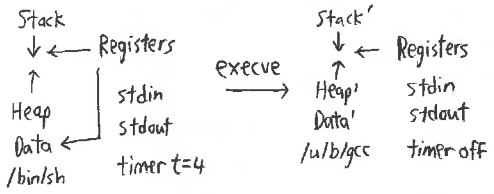
- Defenses
  - Suspend/mask/defer interrupt - but avoid blocking all or too long
  - Lock-free programming if applicable
- Example interrupt, timer - avoid CPU hogs, timekeeping

L4/ Process



Process Creation

- Processes have a lot of state – memory contents, CPU register contents, I/O ports, etc
- fork() – birth by cloning
  - memory : copy all
  - registers : copy all except pid, parent gets child pid, child gets 0
  - file descriptors : copy all references to open file state (not the files themselves!)
  - other stuff : case-by-case 'obvious'
- doesn't create new process! → execve (char *path, char *argv[], char *envp[]) to perform transplant surgery – new memory, new registers, mostly same file descriptors, 'obvious' behavior for other state



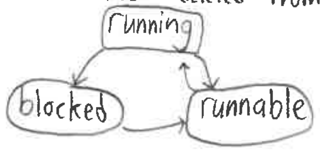
- spawn() – manually specify everything. One benefit is avoiding copy of unused stuff.
- clone() (or Plan9 rfork()) – build new process from old one, specifying what is copied vs what is shared.

Process Setup

- Kernel responsibility
  1. Toss old data, stack
  2. Load executable
  3. Build new stack by transferring argv[] and envp[] to top of new stack, hand-craft ~~main()'s stack frame
  4. Set registers, especially esp ← top of stack frame and eip ← start of ~~main()

Process States

- Running : user mode or kernel mode
- Blocked : awaiting some event, e.g. I/O completion, message, sleeping. Will not be run by scheduler.
- Runnable :
- Forking : obsolete
- Zombie : has called exit() and parent yet to notice. Parent calls wait() to obtain exit code, after which zombie's PCB is deleted from kernel. (see below for PCB)



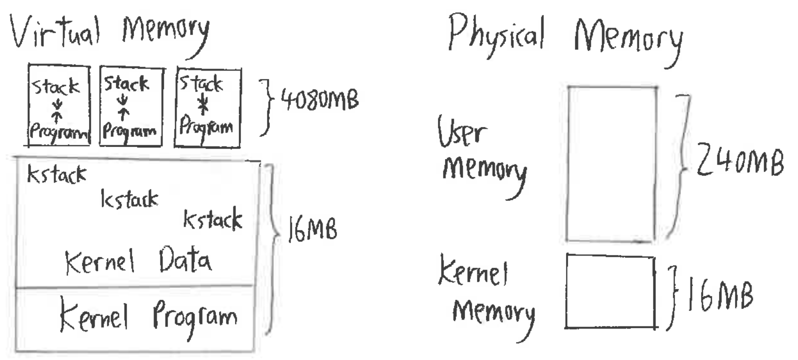
Process Death

- Voluntary via exit()
- Hardware exception, e.g. SIGSEGV
- Software exception, e.g. SIGXCPU
- kill(pid, sig) – deliver sig to process pid

Process Control Block (PCB)

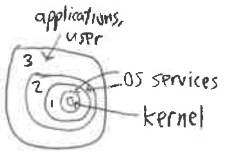
- Contains everything without a user-visible memory address
  - Kernel management information
  - Scheduler state
  - Process number, parent process number, etc

Memory Layout in 410



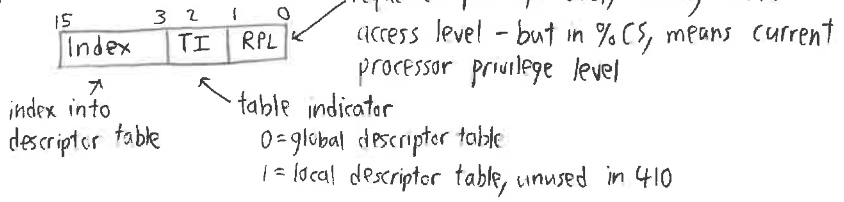
L5/P1

x86 details

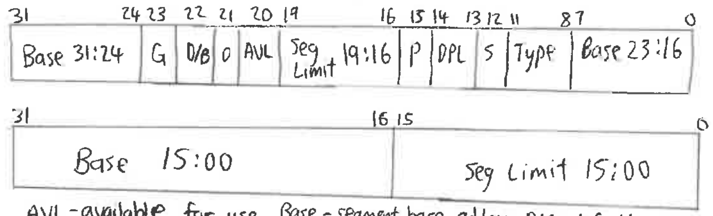


- Privilege Levels (PLs)
  - Processor has 4 privilege levels
  - Zero most-privileged, three least
  - Processor always at one of the four privilege levels at any given time
  - PLs protect privileged data, trigger general protection faults
  - Interrupts and exceptions usually 3→0, sometimes 0→0
  - Running user code requires 0→3
- Memory Segmentation
  - Memory segment = memory range of the same kind
  - Hardware Kinds
    - Read-only memory for boot
    - Unbuffered video memory
  - Software Kinds
    - Read-only code pages
    - Stack
    - Heap
  - In Win16 and Win32, each DLL is multiple segments (not Win64)
  - Mandatory x86 segments: stack, code, data
  - Segments interact with privilege levels, e.g. kernel stack vs user stack, kernel code vs user code
- Segments
  - segment register = cs, ss, ds, ..., gs
  - segment selector
    - which segment table and index?
    - what segment access privilege?
  - segment descriptor
    - which memory range?
    - memory range properties?
  - Processor instruction fetch from %CS:%EIP
    - e.g. if eip=0xface, obtain oxface'th byte of code segment

Segment Selectors


15
3
2
1
0
Index
TI
RPL
requested privilege level, usually means access level - but in %CS, means current processor privilege level
index into descriptor table
table indicator
0 = global descriptor table
1 = local descriptor table, unused in 410

Segment Table Entry Layout



AVL - available for use, Base - segment base address, D/B - default operation size
0 = 16-bit segment
1 = 32-bit segment
G - granularity, P - segment present,
DPL - descriptor privilege level, type - segment type,
S - descriptor type (0 = system, 1 = code or data)

Segment selector index + segment base

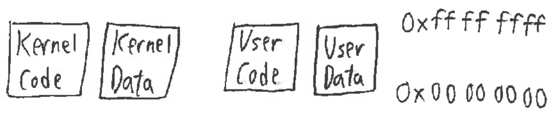
$= \text{linear virtual address} \xrightarrow{\text{virtual memory}} \text{physical address}$

Implied Segment Registers

- CS - fetch code, all instructions fetched from %CS:%EIP
- SS - stack segment, push and pop use %SS:%ESP
- DS - default segment for data access
  - mov (%eax), %ebx fetches from %DS:%EAX
  - but can specify ES, FS, GS instead

More on Segments

- Need not be fully backed by physical memory, can overlap
- Segment cannot be RWX, only RX or RW
- In 410,



Continued,

L5/P1 cont.

Instruction Execution, Processor POV

- Regular work - this, then next
- Branch - this, then somewhere else
- Surprise - must suddenly run a different body of code

Surprises

- Exception - an instruction broke
  - SIGSEGV, page fault, division by 0, illegal instruction...
  - Either fix and rerun, or kill program
- Trap - an instruction asks for help
  - syscall, invoke kernel to do X
  - resume at instruction after trap
- Interrupt - I/O device needs attention
  - defer random instruction while driver runs
  - resume at deferred instruction

Getting Kernel Attention

- Device - hardware interrupt, more later
- User processes - "software interrupt", i.e. not an interrupt, Intel's INT n

Programmable Interrupt Controller (PIC)

- PIC serializes and delivers interrupts

$$\begin{tikzcd}
CPU & PIC\ 1 \arrow[l] & PIC\ 2 \arrow[l] & \\
timer \arrow[ur] & keyboard \arrow[u] & IDE1 \arrow[u] & IDE2 \arrow[ul]
\end{tikzcd}$$

To Processor ←

PIC 1	PIC 2
0 Timer	0 Real time clock
1 Keyboard	1 General I/O
2 Second PIC ←	2 General I/O
3 com2	3 General I/O
4 com1	4 General I/O
5 LPT2	5 coprocessor
6 Floppy	6 IDE bus
7 LPT1	7 IDE bus

Interrupt Handshakes

Processor		Device
	← assert interrupt	Full. Assert interrupt, don't deassert until processor explicitly dismisses.
Invoke interrupt handler.	request data →	Send data, reducing fullness.
Process or enqueue data.	← receive data	
Dismiss the interrupt	dismiss signal →	Stop asserting interrupt, ready to interrupt again.

- PIC automatically defers new interrupts from a device until the old one has been dismissed by processor.

Interrupt Descriptor Table (IDT)

- IDT entry = function pointer + flags

31–16	15	14–13	12–8	7–5	4–0
Offset 31:16	P	DPL	0 D 1 1 1	0 0 0	reserved

31–16	15–0
Segment Selector	Offset 15:0

- Notable IDT entries
  - 0, divide by zero
  - 14, page fault
  - 32, keyboard interrupt

Back to Surprises

- Sync or async?
  - sync → because of instruction, cannot be deferred
  - async → random time, can be deferred
- What next?
  - retry (exception)
  - kill (exception)
  - run next instruction (trap, interrupt)

Device Communication

- eg cursor → I/O Ports - inb(), outb(), NOT memory! Beware trying to memcpy.
- eg video → Memory-mapped I/O - magic memory tied to devices
- Drivers normally have top halves and bottom halves
  - top halves ↑ process queued work at a more convenient time
  - bottom halves ↑ interrupt driven executes quickly queues work

L6b / Threads

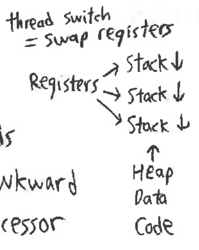
Thread = schedulable register set

Why threads?

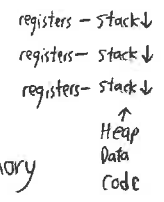
- Shared access to data structures
- Responsiveness, e.g. clean cancellation
- Multiprocessor speedup

Thread Types

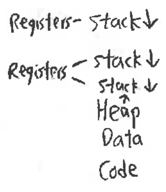
- Userspace N:1
  - + no change to OS
  - - syscall blocks all threads
  - - cooperative scheduling awkward
  - - no benefit on multiprocessor



- Pure kernel threads 1:1
  - + faster on multiprocessor
  - + CPU hogs made to behave
  - - rewrite userspace libraries to be thread safe
  - - requires more kernel memory
    - 1 PCB → 1 TCB + N + CB
    - 1 k-stack → N k-stack



- Many-to-many M:N
  - M user threads share N kernel threads
  - Sharing can be dedicated (user thread 12 owns kernel thread 1) or shared or more



Thread Cancellation

- Async / immediate - stop now, 0 more user space instructions, free stack and registers and vanish
  - - very hard to garbage collect, maintain data structures
- Deferred - threads must check for cancellation or define safe cancellation points

Race Conditions

Process 0	Process 1
ln -s /bin/lpr /tmp/lpr	
	run /tmp/lpr, setuid printer /bin/sh /tmp/lpr ...
rm /tmp/lpr ln -s exploit /tmp/lpr	

!

L7b/Synch

Memory Model

- Note that processors usually queue memory stores and coalesce redundant writes
- Memory barriers are available to stall the processor until the write pipe is empty
- In 410, assume simple model

Synchronization Fundamentals

- Atomic instruction sequence
- Voluntary descheduling

Atomic Instruction Sequence

- Short sequence of instructions (so it is ok to make competitors wait) that must be interleaved with itself
- Low probability of collision
    - Must not use expensive anti-collision
    - Common non-colliding case must be fast

Voluntary Descheduling

- Anti-atomic, want to be maximally interleaved against
- Rely on OS CPU descheduling

OS Naming Convention

-
  ```
  do {
      entry section:
      critical section:
      exit section:
      remainder section:
  } while (1);
  ```

- For 2-process protocols, assume:
    - Multiple threads
    - Shared memory, no locking/atomic instructions
    - No thread runs at 0 speed
    - Thread i = us, thread j = other
    - i, j are thread-local variables $\{i,j\}=\{0,1\}$, $j=1-i$

Critical Section Requirements

- Mutual exclusion

  At most one thread is executing each critical section.
- Progress

  Choosing protocol must have bounded time.

  Common bug: choosing next entrant cannot wait for non-participants.
- Bounded Waiting

  Cannot wait forever after starting entry protocol, bounded number of entries by others.

  (Note. Not necessarily a bounded number of instructions)

Peterson's Solution 1981

- Take turns when necessary

```
boolean want[2] = {false, false};
int turn = 0;
want[i] = true;
turn = j;
while (want[j] && turn==j) { continue; }
Ⓐ /* CRITICAL SECTION */
want[i] = false;
```

- Proof of mutual exclusion. Suppose two threads in Ⓐ, then want[i]==want[j]==true. So exited by turn, but turn cannot be both 0 and 1, ↯.

Lamport's Bakery Algorithm

- Bakery counter, (ticket number, process number) tuple
- Phase 1. Pick max$\left\{\begin{matrix}\text{current}\\\text{numbers}\end{matrix}\right\}$ +1 as your number.
- Phase 2. Your turn when holding lowest (ticket id, pid)

Phase 1.

```
choosing[i] = true;
num[i] = max(num[0], ...) + 1;
choosing[i] = false;
```

Phase 2.

```
for (j=0; j<n; ++j) {
    while (choosing[j]) { continue; }
    while (num[j] != 0
           && (num[i], i) > (num[j], j)) { continue; }
}
/* CRITICAL SECTION */
num[i] = 0;
```

L8b/Synch

Mutex

- Also known as lock or latch
- Intel's atomic XCHG instruction

```
int 32  xchg (int32 *lock, int32 val) {
    register int old,
    old = *lock;  } bus is locked
    *lock = val;  }
    return old;
}
```

- init: lock_available = 1
- try-lock: won = xchg (&lock_available, 0);
- spin-lock: while (!xchg (&lock_available, 0)) {continue;}
- unlock: EXPECT_EQ (0, xchg (&lock_available, 1))
- Problem: guarantees mutual exclusion and progress, but not bounded waits
  - Fairness via lock()
  - Fairness via unlock(), waiting [j] = false or set lock available
- Note that unlocker may be required to use special memory access (atomic release vs normal memory write)
- Note above fairness protocol sucks
  - Size of thread population required
  - O(n) in max possible competitors
- Metrics should consider typical access pattern and runtime Environment
  - Uniprocessor
    lock()'s xchg is wasteful, yield to the lock holder instead
    unlock() potentially an unfair competition, depends on OS scheduler
  - Multiprocessor
    Spin-waiting OK since short critical sections, low contention.
    Next xchg winner depends on memory hardware.

Load-Linked / Store-Conditional (LL/SC)

- LL (addr) fetches old value from memory
- SC (addr, val) stores val to addr only if nothing else has stored to that address in between, else fails
- Implemented by cache snooping on the shared memory bus
- Inherently nondeterministic, kills Mozilla rr

Intel i860 Magic Lock Bit

- Instruction locks bus, disables interrupts
- 32-instruction timer triggers exceptions, and exceptions such as page fault or zero divide unlock bus
- Allows implementation of synchronization primitives

Kernel

- The kernel could in principle provide mutexes
  - Detect context switch via instruction, memory addresses, Flag..
  - Handle unusual case by giving time slice, simulating unfinished instructions, rollback..
- In practice, too expensive and too complicated

L9b/Synch

Condition Variables

- Keep track of threads temporarily blocked
- Allow notifier threads to unblock thread(s)
- Must be thread safe
- Be careful with signals

```
cond-wait(cvar, world-mutex){
    lock (cvar→mutex)
    enq (cvar→queue, gettid())
    unlock (world_mutex)
    ATOMIC {
        unlock (cvar→mutex)      ← cond_signal
        deschedule()               could get lost
    }                              Or wake running
    lock (world_mutex)             thread
```

Semaphore

- Counted resource, int represents #available
- wait()/P()/dec() } must be atomic!
- signal()/V()/inc() }

Monitor

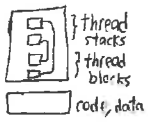
- Invisible compiler-generated object
- Module of high-level language procedures which all access some shared state
- Thread running in any procedure blocks all thread entries
- Basically cvar with implicit monitor mutex used throughout
- Different signal policies, no standard

L11b/ Yield

Pure user-space threads

- Single-threaded process, no thread_fork
- Thread = stack + TCB
- yield (user-thread-i){

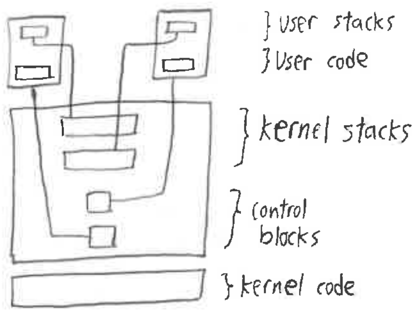
```
    save registers on stack
    tcb→sp = get_esp();
    tcb = find_tcb(user-thread-i)
    set_esp(tcb→sp);
    restore registers from stack
}
```



- User threads share memory, threads are not protected from each other

Kernel Processes

- Do not (usually) share memory
- Do not modify other processes' shared registers



P1 : yield (P2)

↓ INT 50

Processor trap protocol stack switches (user→kernel), saves some registers to P1's kernel stack

Top of kernel stack: esp0

x86 trap frame: ss, esp, eflags, cs, eip

↓

Assembly wrapper saves more registers, invokes C trap handler

↓

Assembly wrapper restores registers from P1's kernel stack

↓ IRET

Processor return-from-interrupt restores ss, esp, eflags, cs, eip

↓ INT 50 completes

P1 yield() returns

```
int sys_yield (int pid) {
  return process_switch(pid);
}
```

```
process_switch( )
  · ATOMIC {
      enq_tail (runqueue, cur_pcb)
      save P1 registers
      cur_pcb = deq (runqueue, P2)
      stackpointer = cur_pcb→sp
      restore P2 registers
    }
```

Context Switches

- Happens in kernel for multiple reasons
  - yield()
  - $P1 \xrightarrow{message} P2$
  - P1 blocked on I/O, run P2 – when I/O complete, who runs?
  - CPU preempt by clock interrupt
    - Clock interrupts are involuntary, yielding process does not specify who to yield to

L12/ Deadlock

Deadlock

- Set of N processes, each waiting for an event which can only be caused by another process in the set
- Every process will wait forever
- Four requirements for deadlock
  - ① Mutual Exclusion
    Resources are not thread-safe/ re-entrant and must be allocated to one owner at a time, cannot be shared.
  - ② Hold and wait
    Have some resources, wait for more
  - ③ No Preemption
    Can't force process to give up resource
  - ④ Circular Wait
    Cycle in resource graph
- Four solutions
  - Prevention - prevent one of 4 requirements
  - Avoidance - predeclare usage
  - Detection / Recovery - abort victim
  - Reboot when 'quiet' - windows :)

Deadlock Prevention

- Ban mutual exclusion?
  - Many resources need it
- Ban hold and wait?
  - Acquire all-or-none
  - May result in starvation since it is harder to get more resources
  - Low utilization
- Ban non-preemption?
  - Some resources cannot be cleanly preempted, e.g. CD burner
- Ban circular wait?
  - Impose total order on resource acquisition aka lock order
  - May not be possible

Deadlock is not

- Synchronization bug
  - Remains after sync bugs are resolved
  - A resource usage design problem
- Starvation
  - Both never get resources
  - But deadlock ≈ progress, starvation ≈ bounded wait
- Livelock
  - Livelock is continuous change of state without making progress

L13 | Deadlock

Deadlock Avoidance

- Processes predeclare usage patterns or at least maximal resource usage
- Processes proceed to completion
- $(P_1, P_2, \ldots, P_n)$ is a safe sequence if every P_i satisfiable by currently free resources F and resources currently held by $P_1, P_2, \ldots, P_i$
- System is in a safe state if at least one safe sequence exists
- Request manager grants requests iff enough resources free now and enough resources would still be free afterwards. Otherwise tells the requesting process to wait.
- Note that unsafe may not be necessarily fatal - process may exit early, not use max resources.

Deadlock Detection

- DB-style, scan graph periodically
- Think of scan policies
- When deadlock found, either
  1. Abort → but rerunning processes expensive, long tasks starve
  2. Preempt → but who? beware starvation.
     Tell process to give up some resources and try again later via EDEADLOCK.

Banker's Algorithm

```
int cash;
int limit[N], out[N];
boolean done[N];
int future;

int progress(int cash) {
    for (i=0; i<N; i++)
        if (!done[i])                          } can cover
            if (cash >= limit[i]-out[i])       } existing
                return i                       } customers
    return -1
}
boolean is_safe(void) {
    future = cash
    done[0..N] = false
    while ((p=progress(future))>0) {    } cover customers,
        future += out[p]                } update balance as
        done[p] = true                  } we can recover all
    }                                     money
    return done[0..N]==true } could recover money and cover
}                             from all customers
```

L3a / Questions

① I don't get ___.

- I read X and Y, understand Z, how to apply W?
- Spec says ☐, unclear if _ or _ because ___.

② Can I assume ___?

- If I assume ___ and I'm wrong, what happens?
- If I don't assume ___, penalty?

③ Right way to ___?

- Find 2 or 3 ways to ___.

④ Decide between X or Y?

	X	Y
metric 1	good	bad
metric 2	bad	good
Conclusion	This, because..	

L6a / Debugging

Measurement Techniques

- printf()
- single-stepping
- breakpoint, watchpoint
- esp, eip - should always make sense
- cs, ds, ss - not that many legal values
- eflags, cr0 - try harder

L7a / define

- Magic constants → easier change requirement
- Careful use of (), consider multiplicity
- do { foo(); } while (0)

L8a / Errors

- Three kinds of error
  - Hmm → try to resolve
  - That's not right → try to report
  - Uh-oh → try to help dev find problem faster
- Corresponding examples
  - Out of bounds → grow array
  - Out of heap → report, pass buck
  - Impossible condition → crash
- If we're here, what happened?
- Now that we're here, what next?

L9a / include

- C has compilation units (≈ ANSI file)
- Idempotent .h files for module-like C
  - ifndef, define, endif
- .h files should contain no code, instead:
  - types
  - public methods
  - constants
  - sometimes macros

L11a / Lost

- TOCTTOU - time of check to time of use
- But special case, if revoked condition will get unrevoked soon then easy to fix by using while() instead of if()