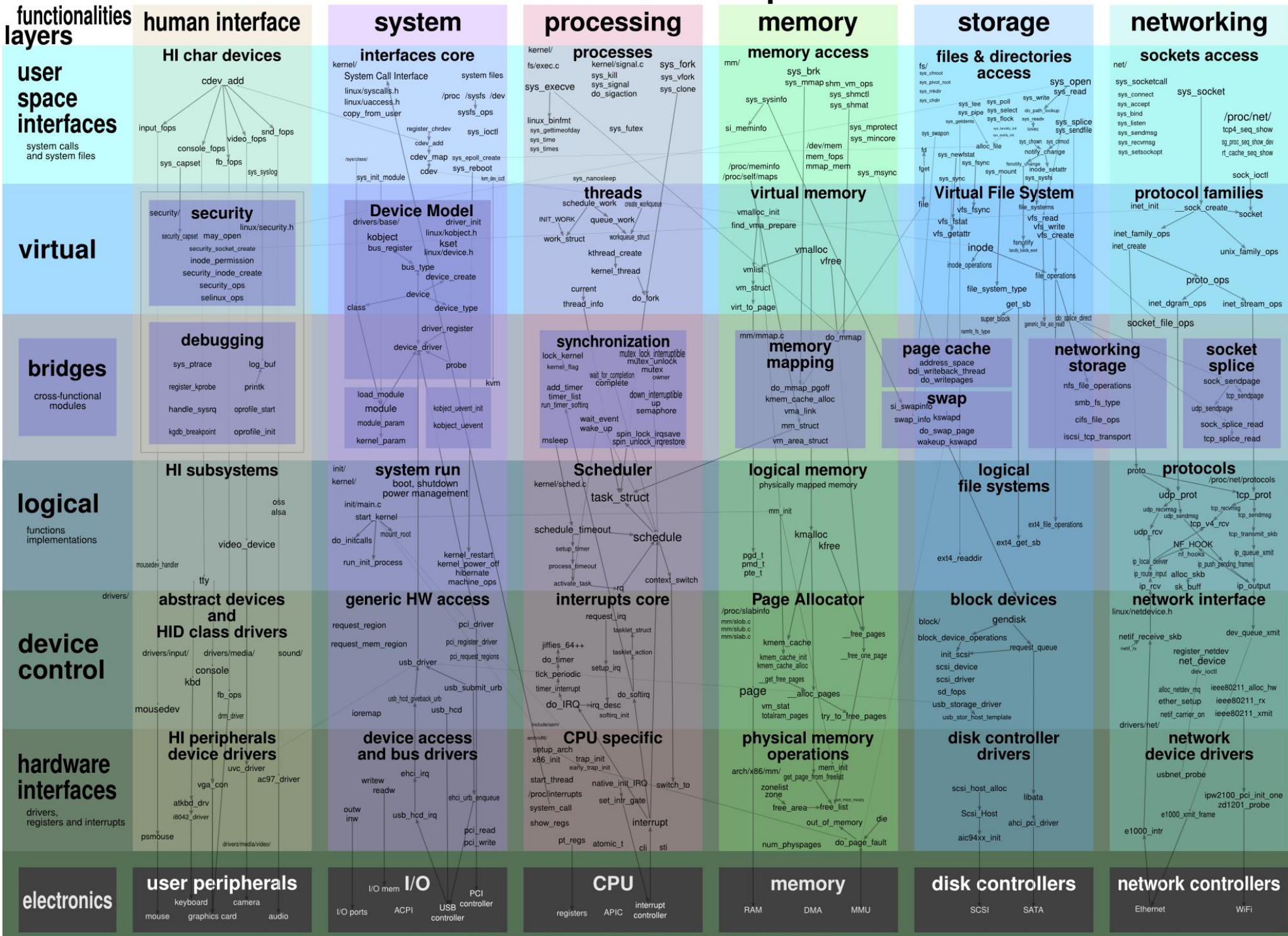
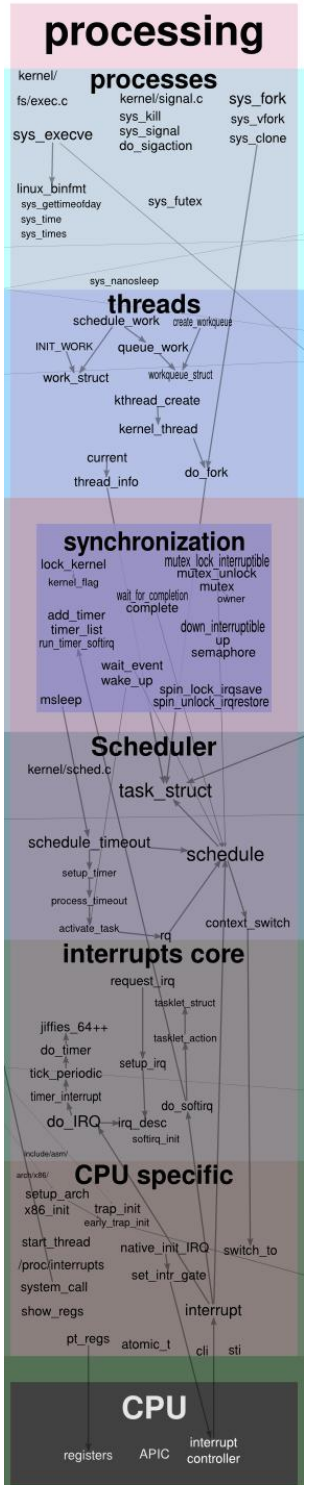
Linux kernel map



© 2007, 2010 Constantine Shulyupin www.MakeLinux.net/kernel_map

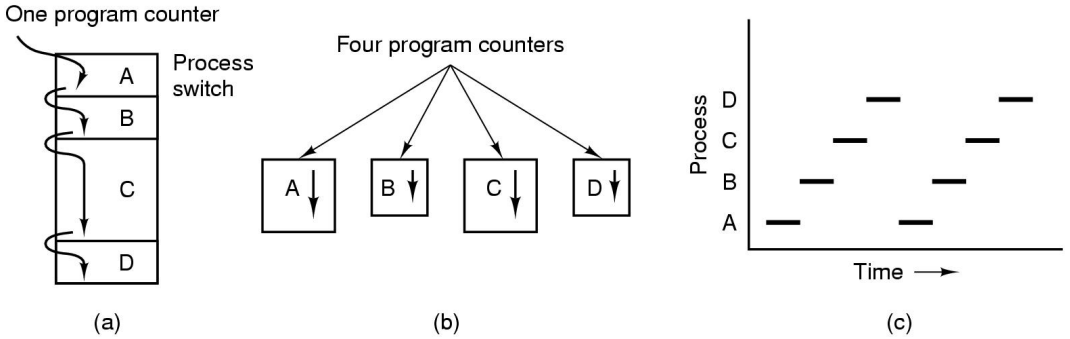
Processes in General

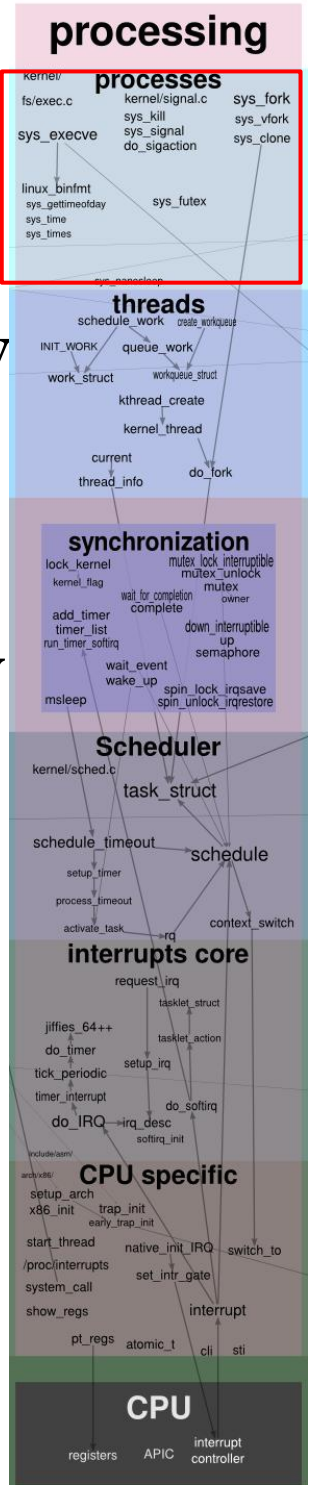
- Multiuser systems allow several processes to be active concurrently
- Systems allowing concurrent active processes are multiprogramming systems
- scheduler decides which process is allowed to progress
  - OS may be nonpreemptive
    - scheduler invoked only when CPU is voluntarily relinquished
  - or preemptive
    - scheduler invoked by OS periodically, a necessity for multiuser systems



Classifying Systems

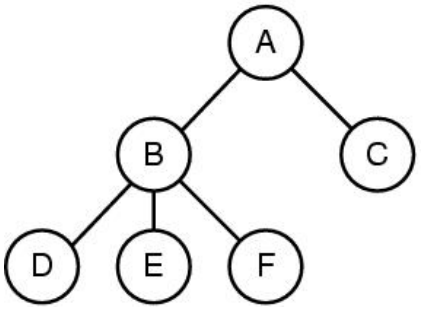
- Uniprogramming: one program resident in memory
  - e.g. DOS
  - Advantage: simple OS
  - Disadvantage: inconvenient, system performs poorly
- Multiprogramming: multiple programs resident
  - Most modern OS
  - Advantage: user convenience and system perform.
  - Disadvantage: complex OS

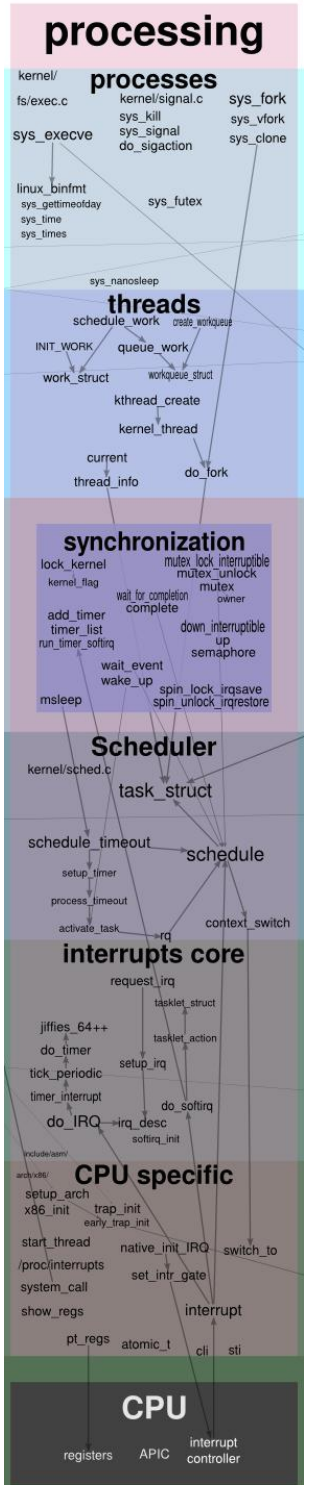




Processes in General

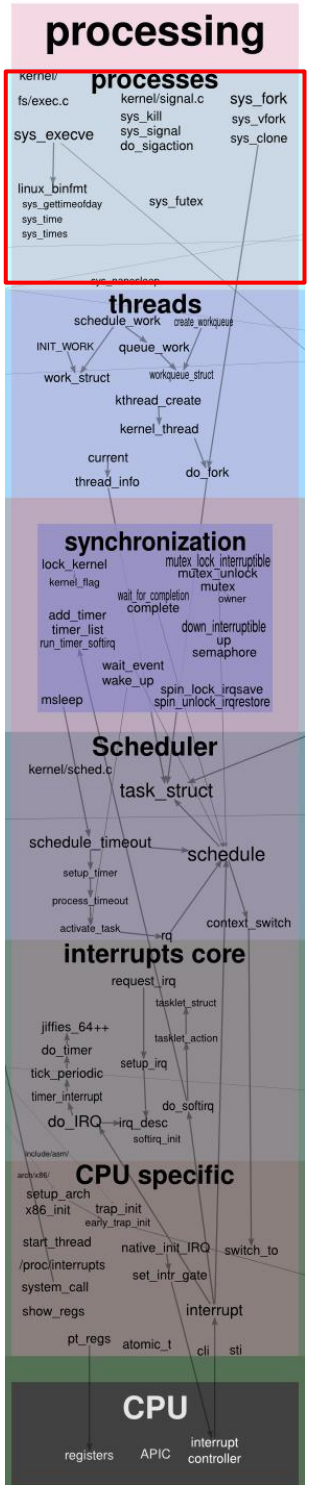
- Processes are programs in execution
- From the OS perspective
  - process = container defining an address space which contains:
    - executable code
    - program data
    - program stack
- OS manages processes
- Processes can reproduce
- Processes can communicate





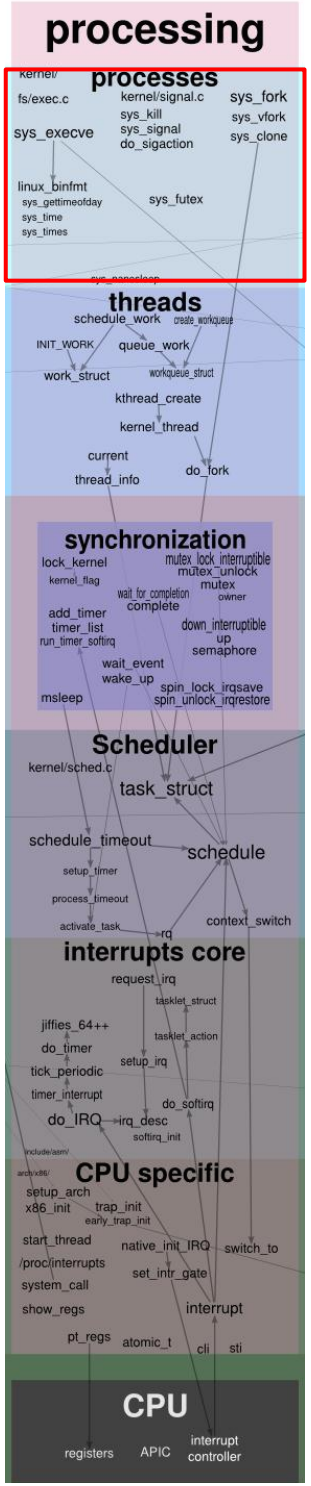
What is a process?

- Process: An execution stream in the context of a process state
- Execution stream
  - Stream of executing instructions
  - Running piece of code
  - Sequential sequence of instructions
  - “thread of control”
- Process state
  - Everything that the running code can affect or be affected by
  - Registers
  - Address space



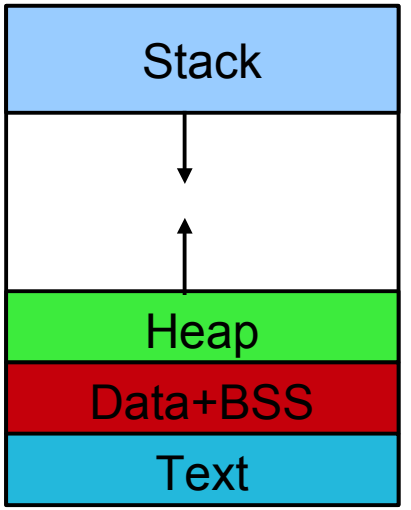
Process vs Program

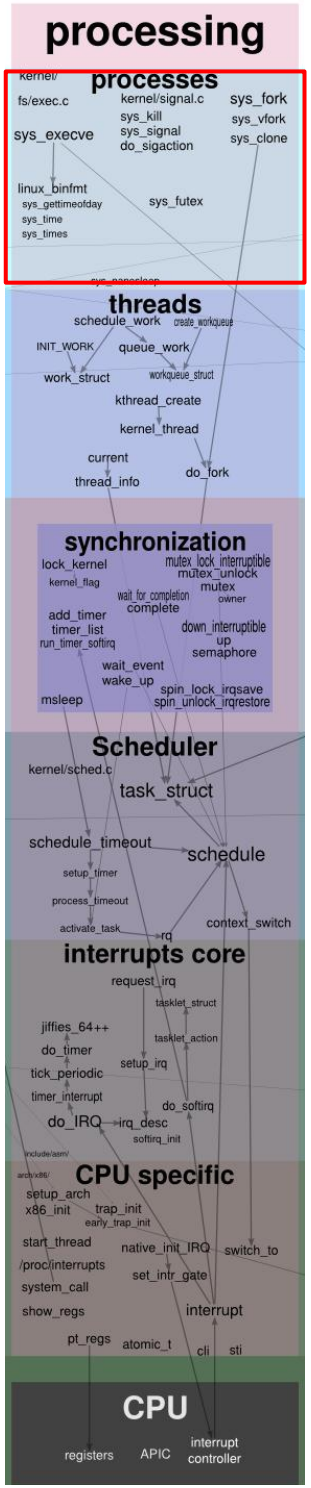
- Process: instance of program in execution
- Program: static code and data
- No one-to-one mapping between processes and programs
  - many instances of same program may be running as processes
  - one program may invoke multiple processes



Process Organization

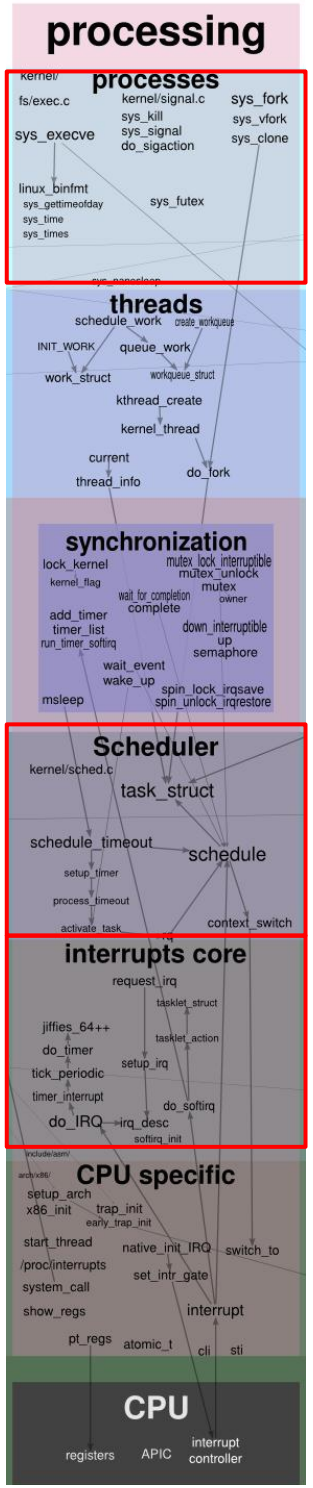
- Each process has own address space
- Memory map:
  - Text: compiled program
  - Data: initialized static data
  - BSS: uninitialized static data
  - Heap: dynamically allocated memory
  - Stack: call stack
- Process context:
  - Program counter (PC)
  - CPU registers



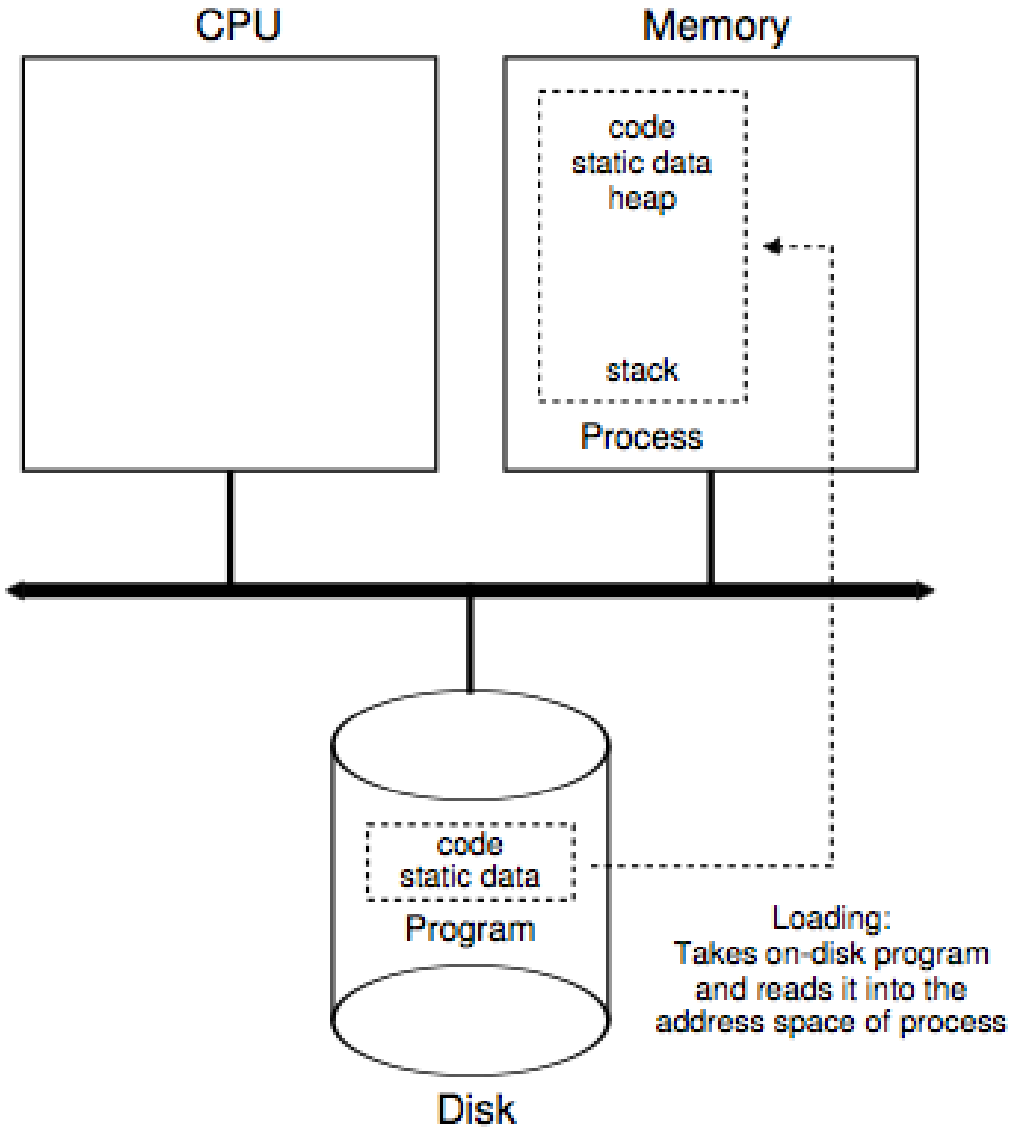


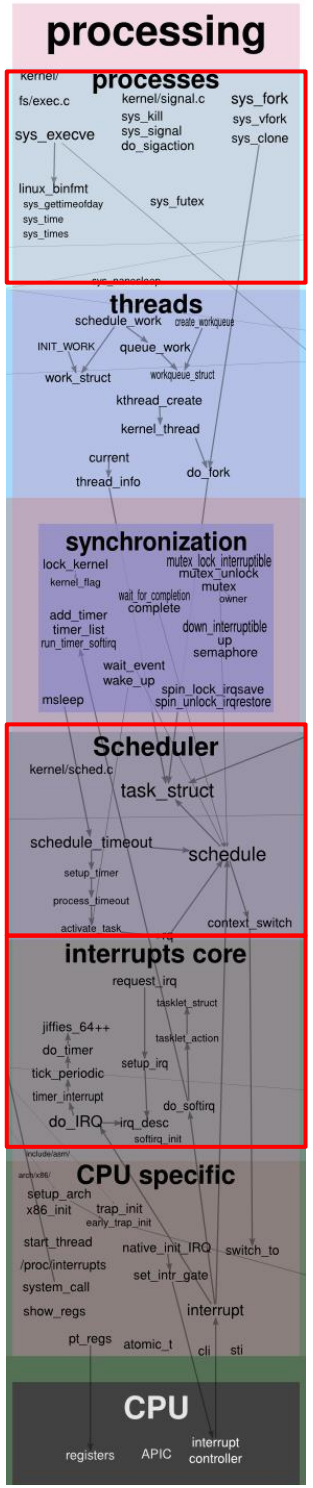
Process Creation

- Option 1: Build new process from scratch
  - Steps
    - load specified code/data to memory
    - create empty call stack
    - create/initialize PCB
    - put process on ready list
- Option 2: Clone existing process and change
  - Steps
    - stop current process and save state
    - copy code/data/stack/PCB
    - add new PCB to ready list



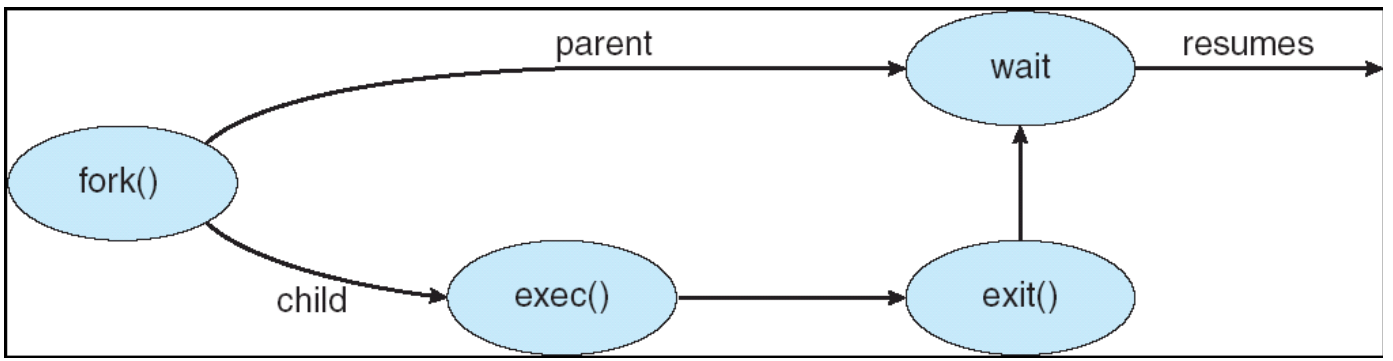
Process Creation

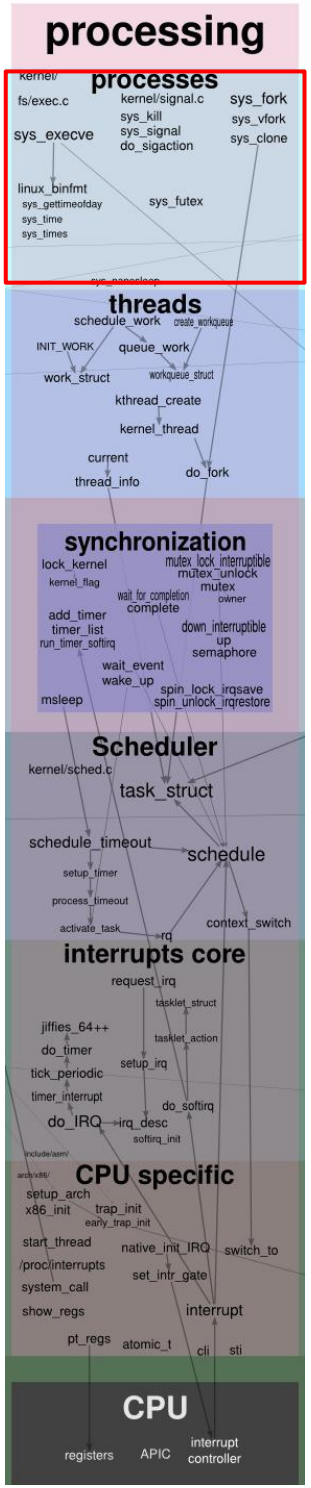




Example: Shell

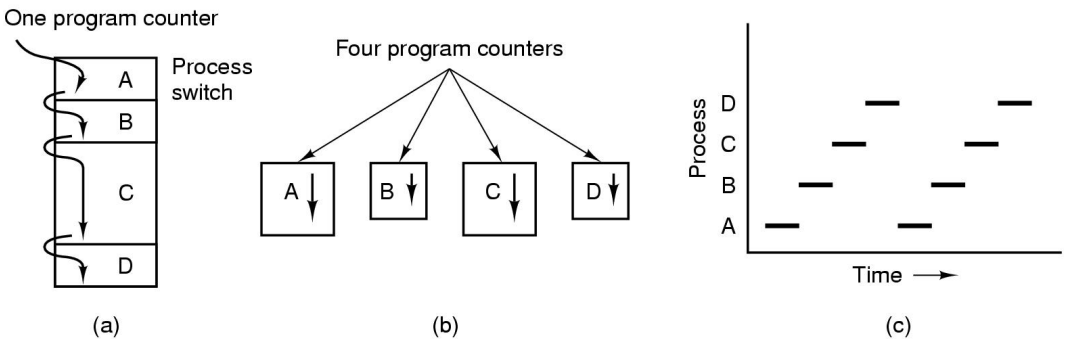
```
while (1) {
      char *cmd = getcmd();
      int retval = fork();
      if (retval == 0) {
         // This is the child process
         // Setup the child's process environment here
         // E.g., where is standard I/O, how to handle signals?
         exec(cmd);
         // exec does not return if it succeeds
         printf("ERROR: Could not execute %s\n", cmd);
         exit(1);
      } else {
         // This is the parent process; Wait for child to finish
         int pid = retval;
         wait(pid);
      }
}
```

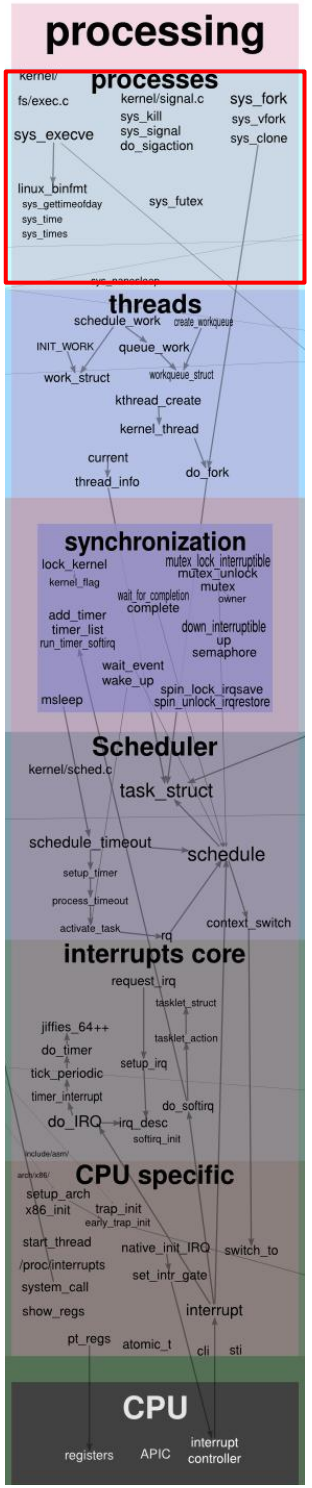




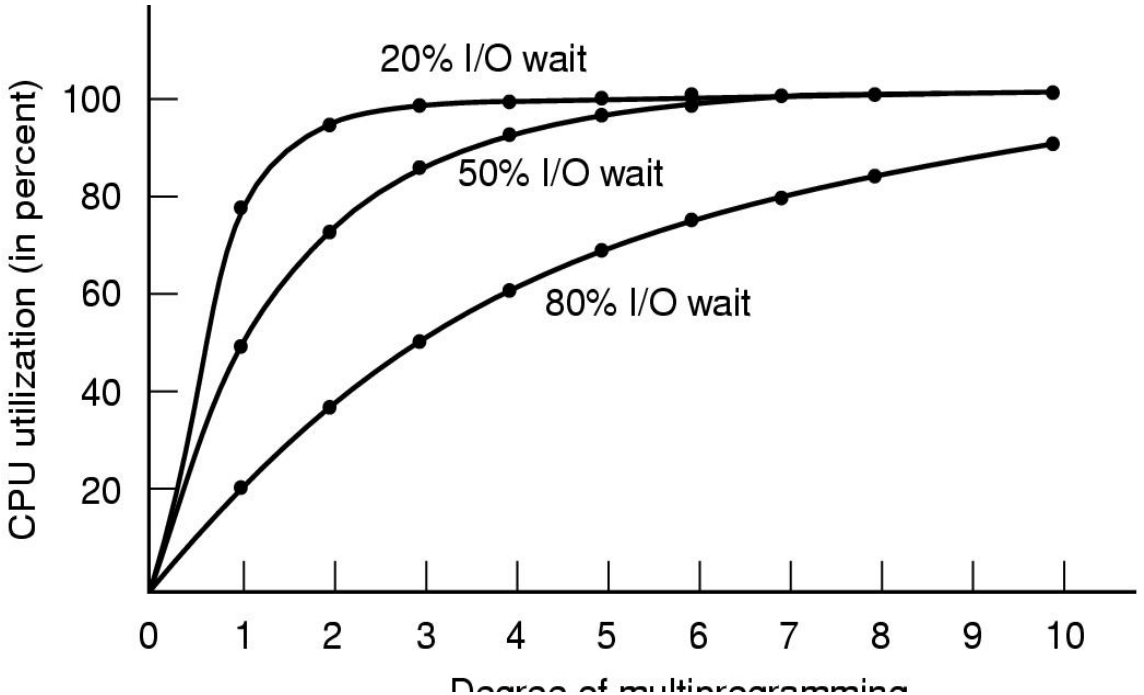
Multiprogramming

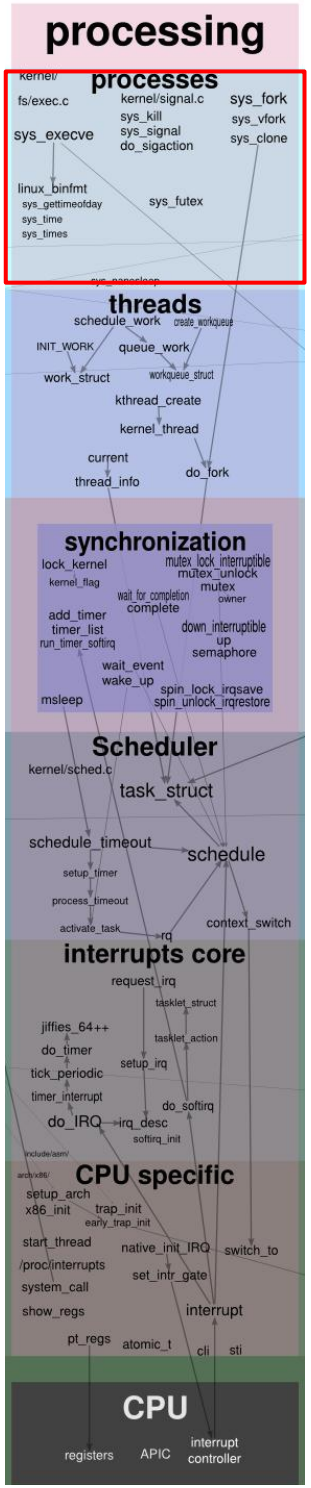
- CPU has one PC
  - must switch between processes
- Internal process state
  - virtualized CPU
- Execution time not uniform
- Process identifiers (PID) unique per process
- Asynchronous events
- Goal: Maximal utilization of CPU





Maximizing CPU Utilization

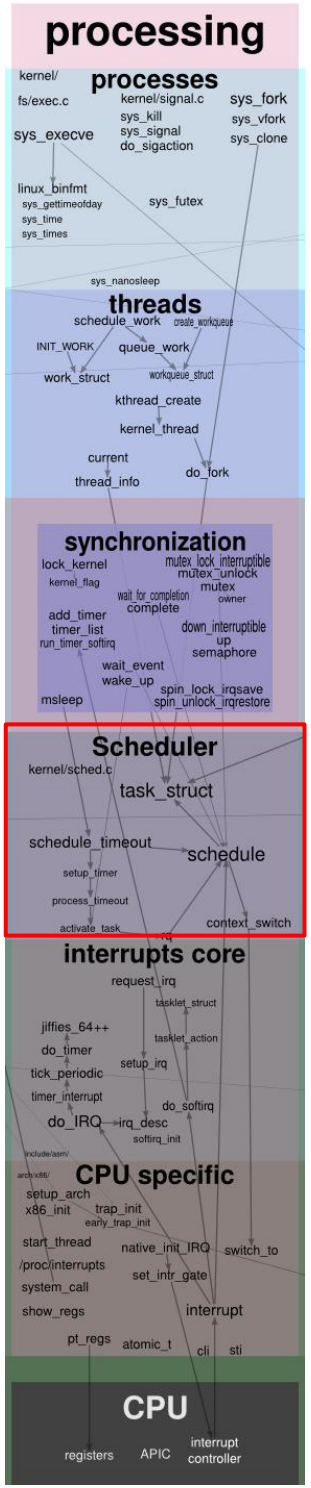




How is it accomplished?

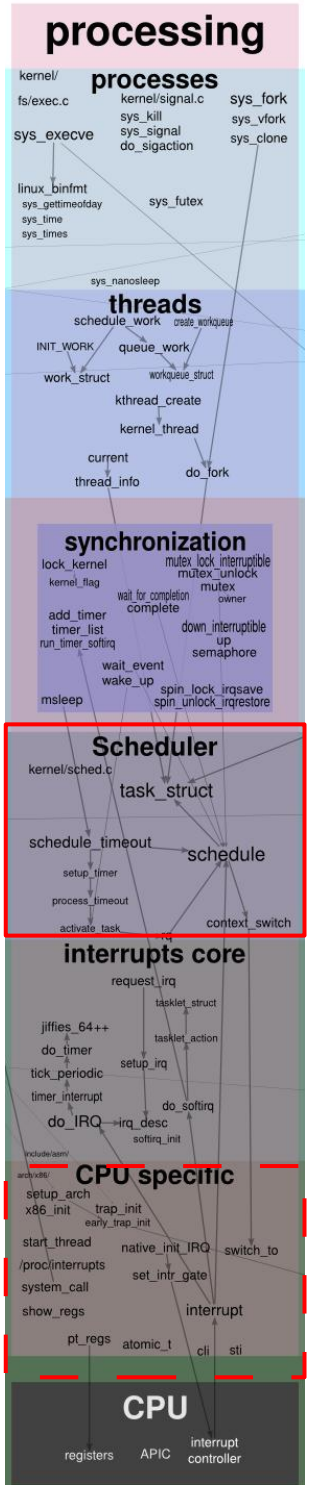
- Scheduler/dispatcher

```
while (1) {
    run process A for some time-slice
    stop process A and save its context
    load context of another process B
}
```



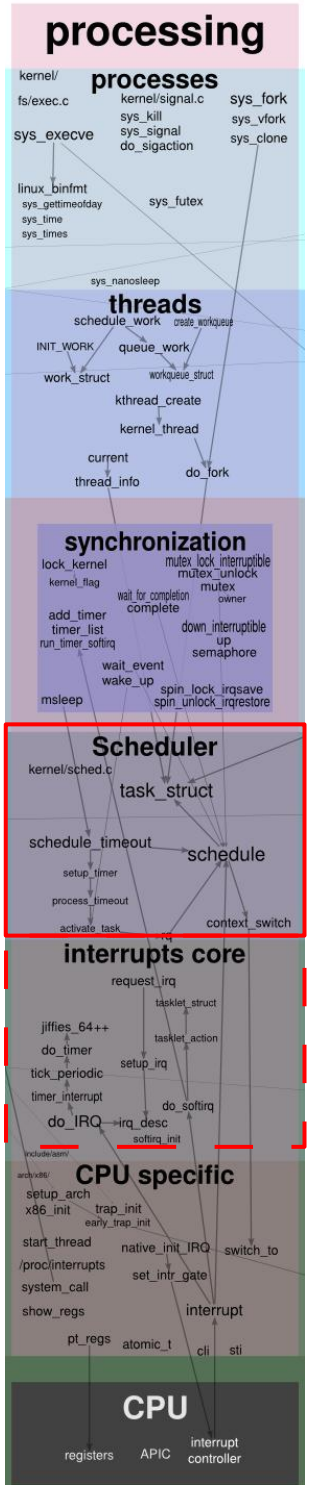
Hardware Support

- First, must differentiate between application and OS
- HW gives us a status word
  - one bit determines user vs system mode
    - system mode: privileged instructions, access to all memory, can change stack pointer…
  - applications run in user mode
  - OS runs in system mode



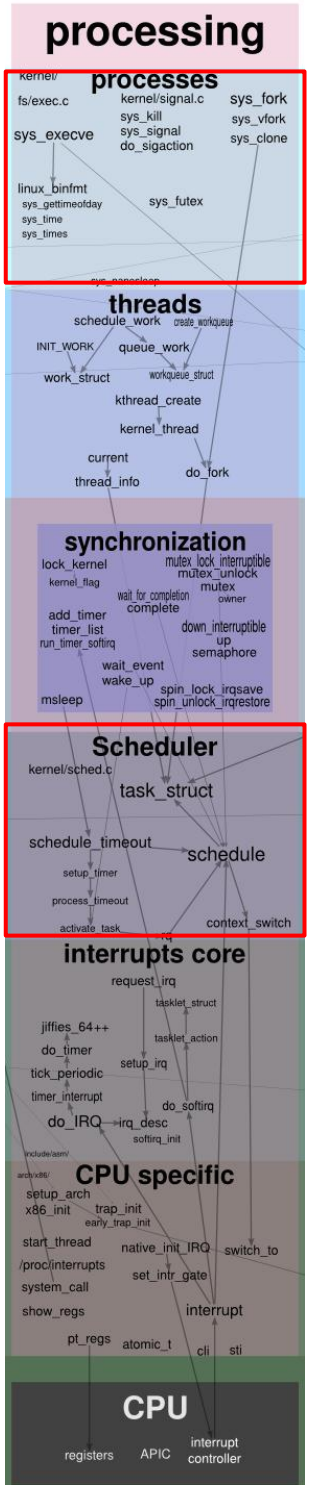
Triggering the OS

- Synchronous interrupts, traps
  - events generated by processes
  - system calls, page faults, etc.
- Asynchronous interrupts
  - events generated by hardware

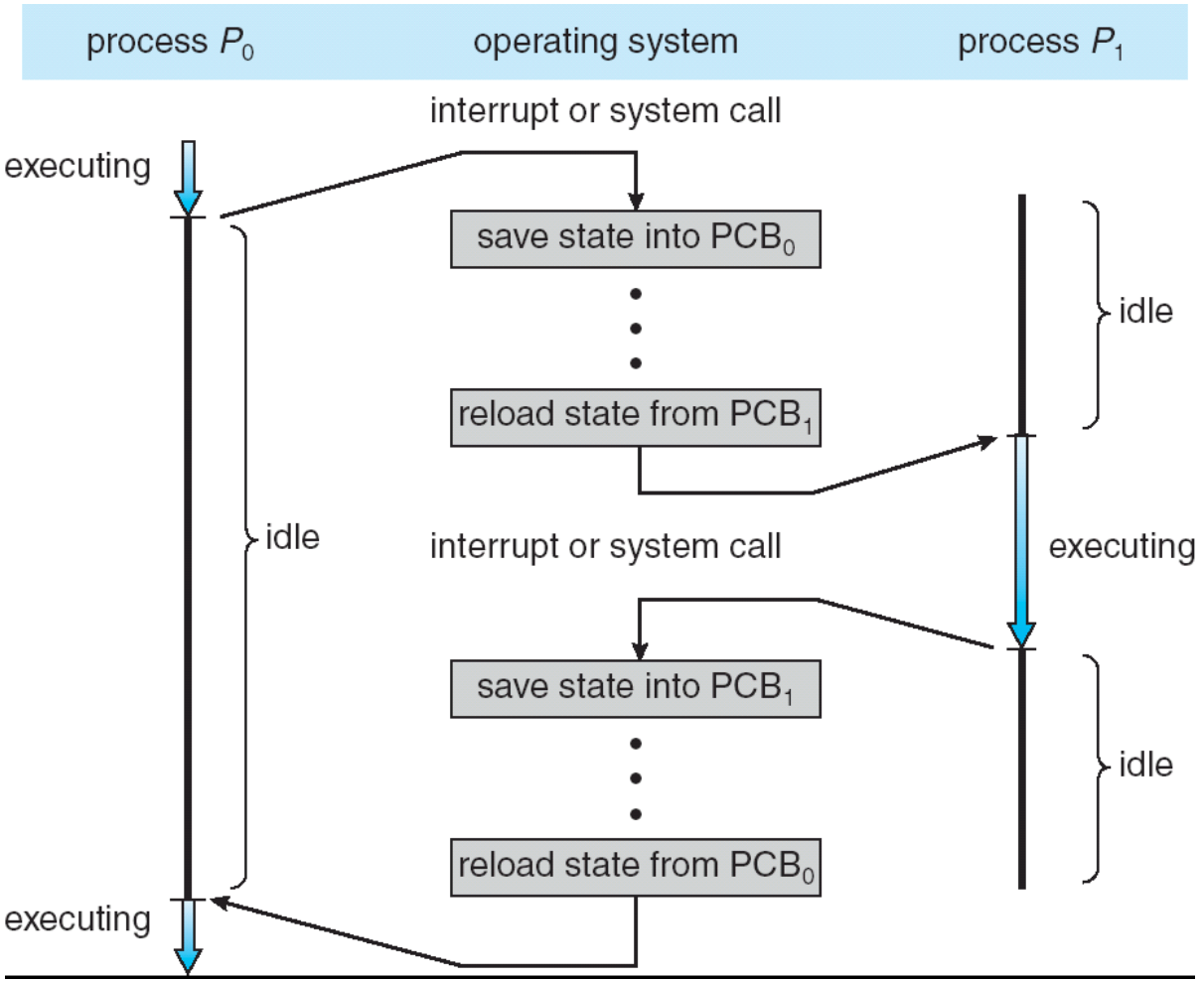


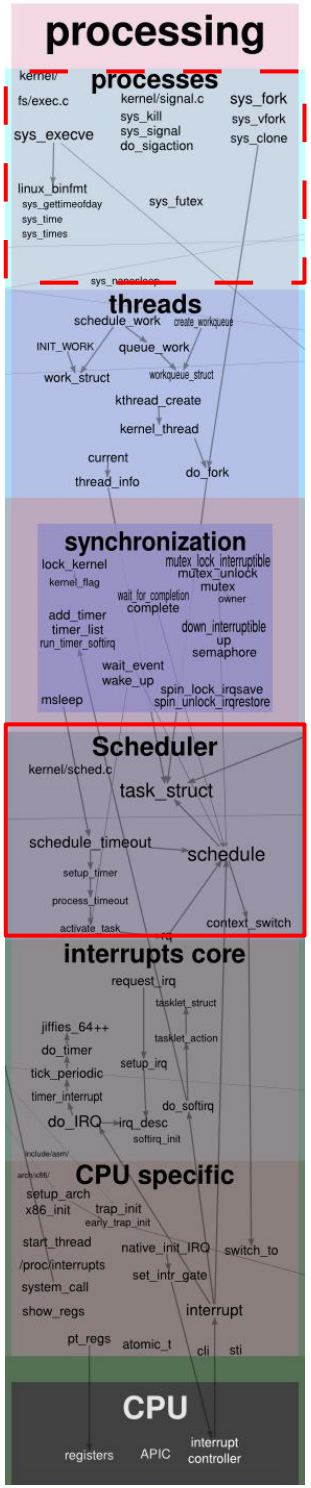
Process Context

- Process Control Block (PCB)
  - PID
  - Process state (running/ready/blocked)
  - Execution state (register values)
  - Scheduling priority
  - Accounting info. (parent/children)
  - Credentials (who owns it, what can it access?)
  - Pointers (open files, etc)
- Let's see an example... DLXOS



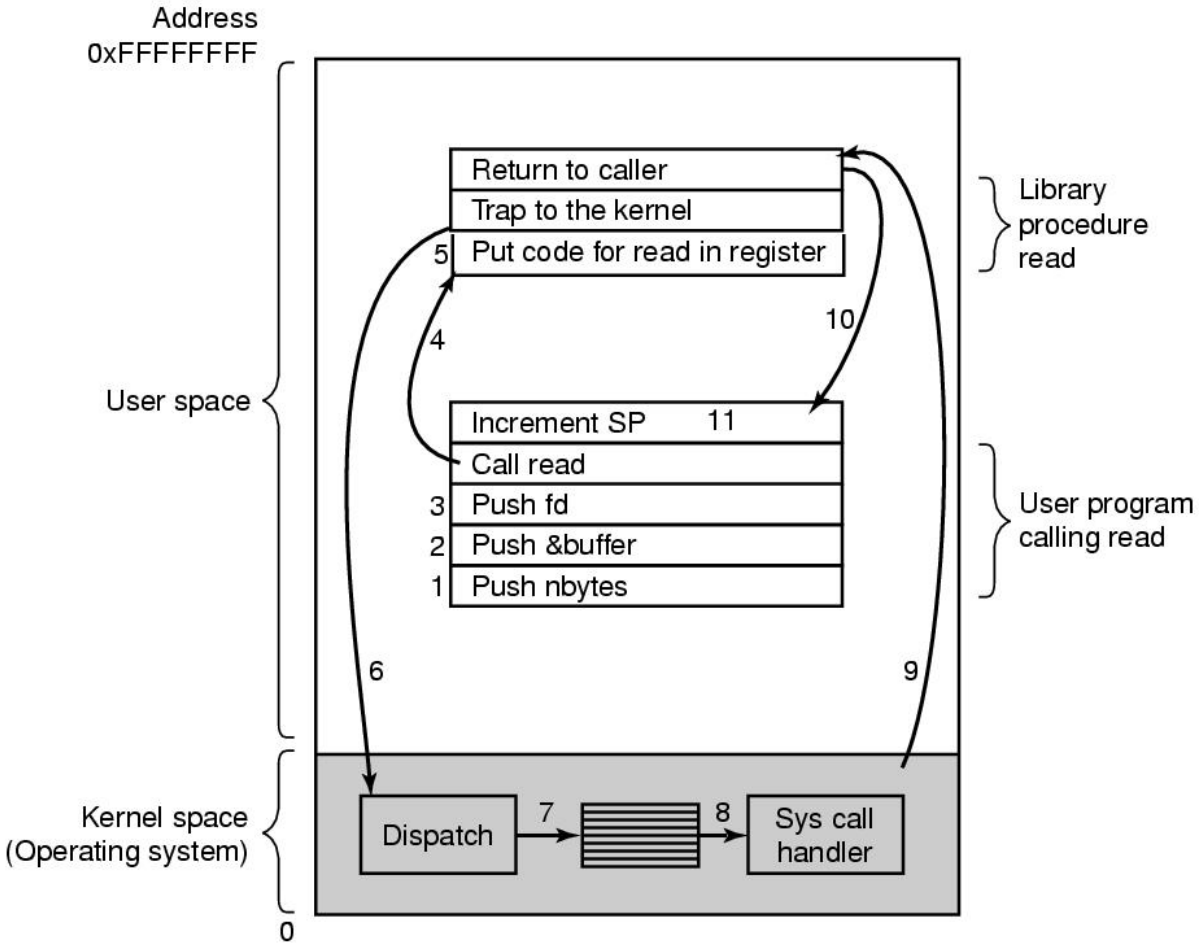
Context Switching

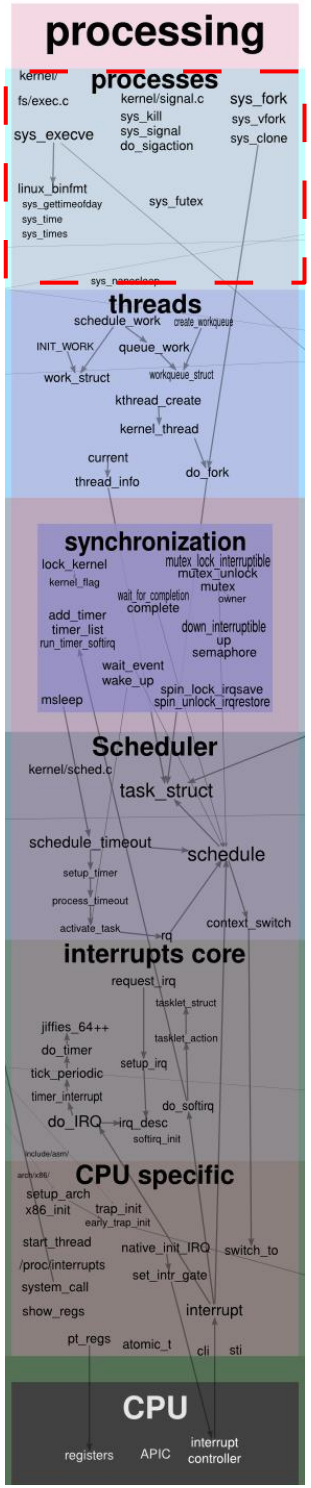




System Call Example

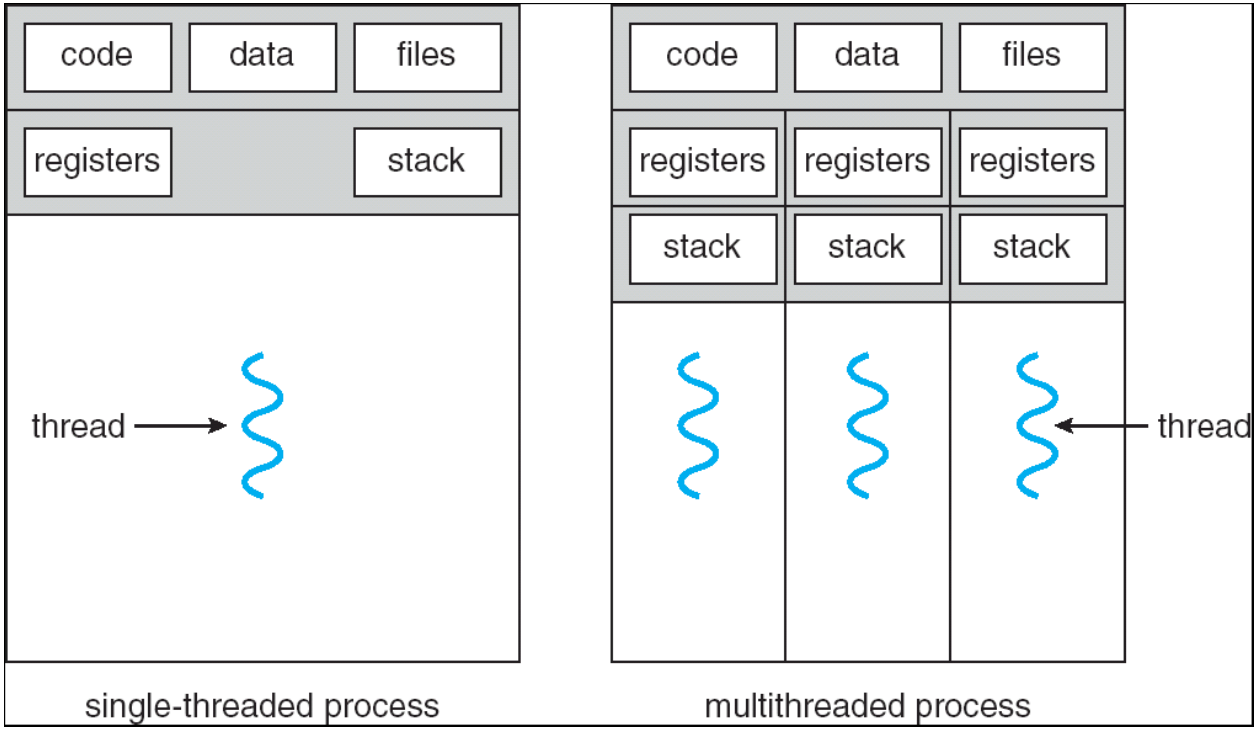
count = read(fd, buffer, nbytes);

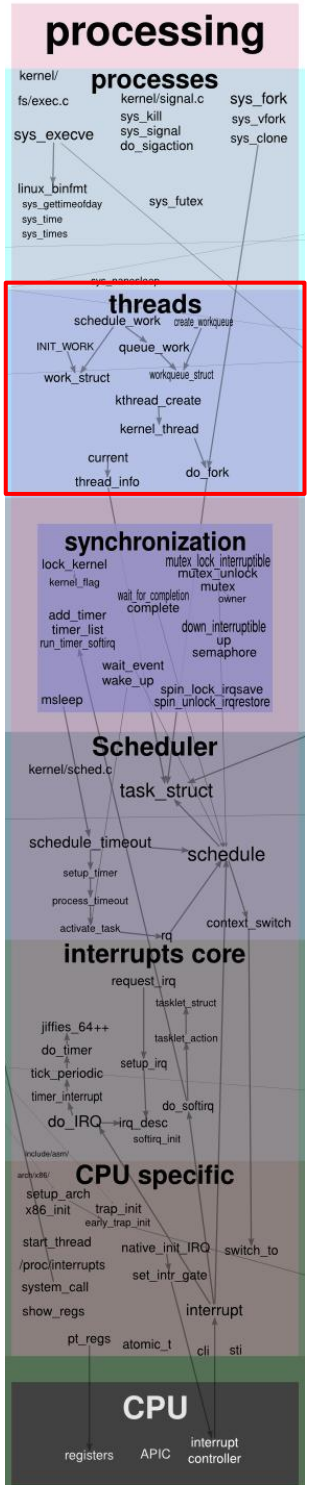




Threads

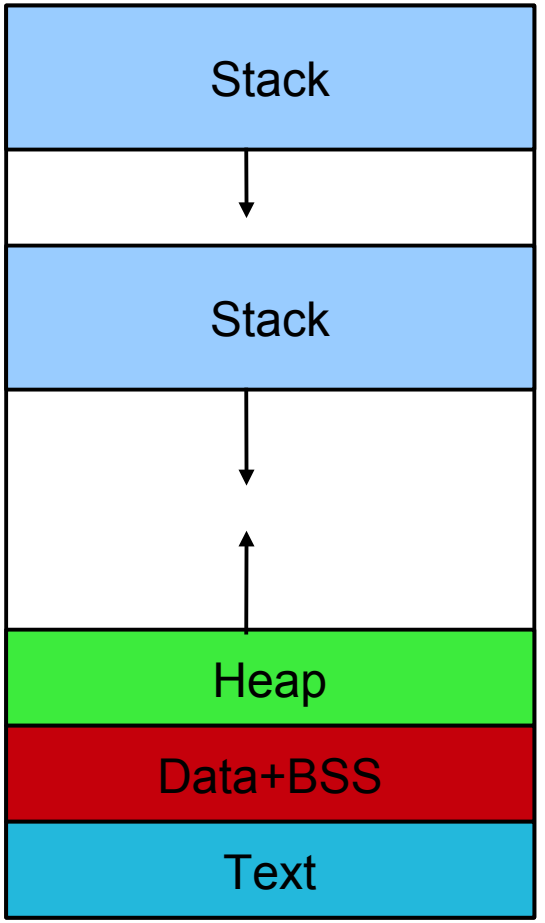
- All threads share memory of parent process
- Have own state
- Variety of implementations exist:
  - win32 threads, c-threads, POSIX threads...

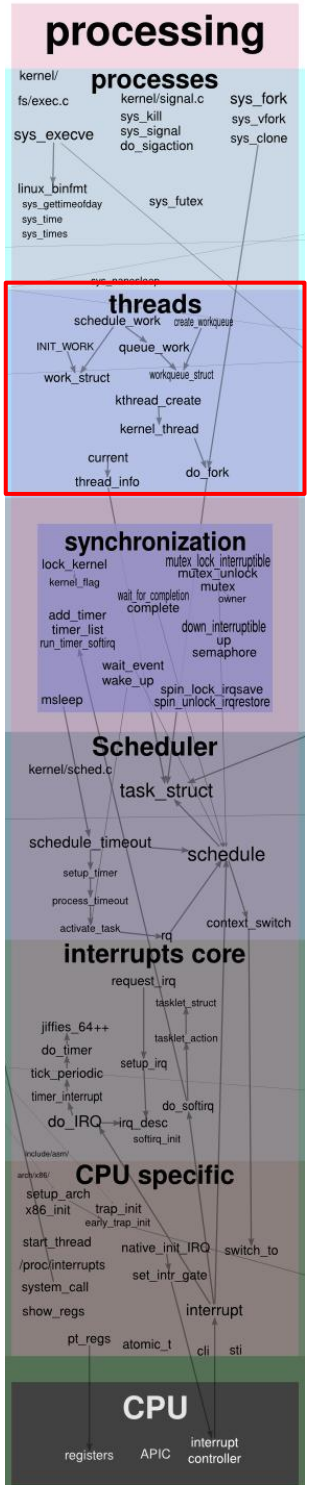




Threads

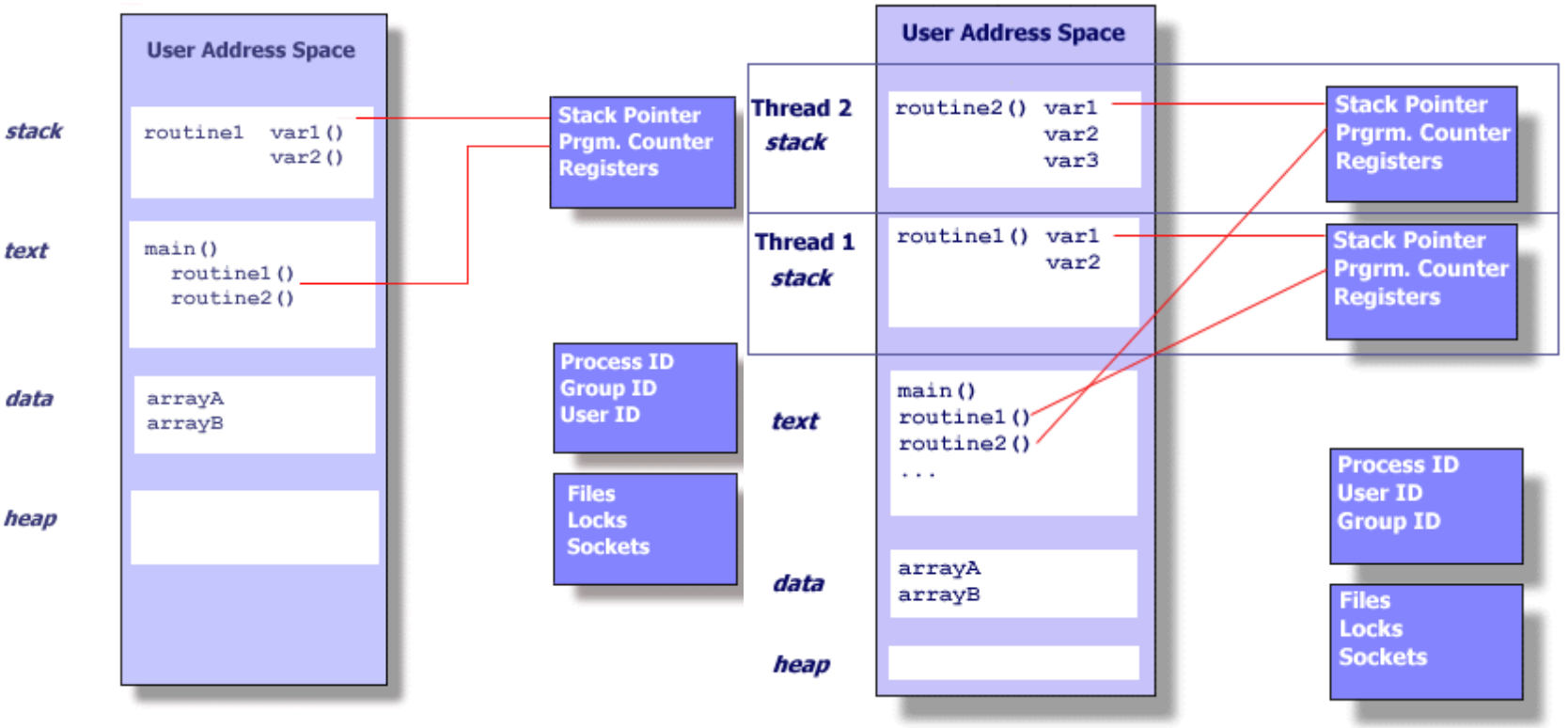
- Process memory map might look like this
- Shared stuff:
  - text, data, bss, heap, working dir., signals, fd's, user/group id's
- Unique to each thread:
  - thread ID, stack, signal mask, priority, registers, stack ptr, instruction ptr.

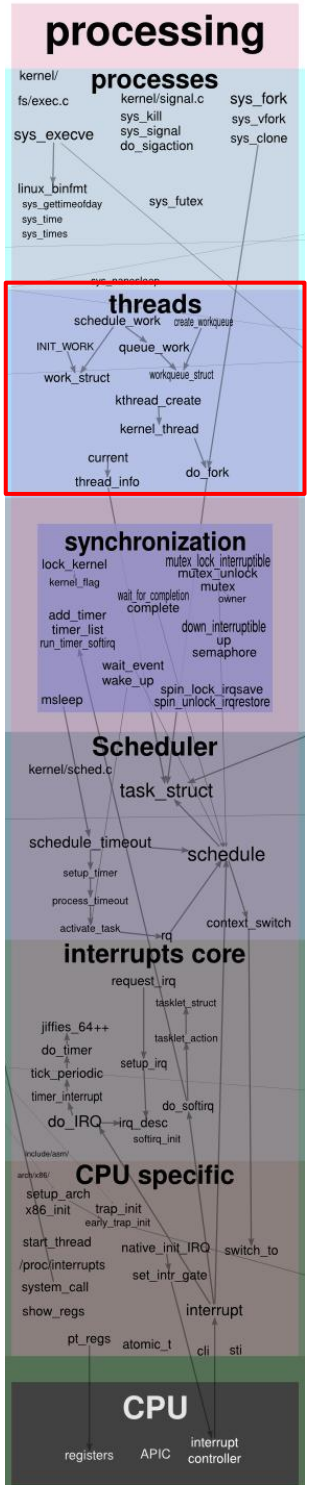




Threads

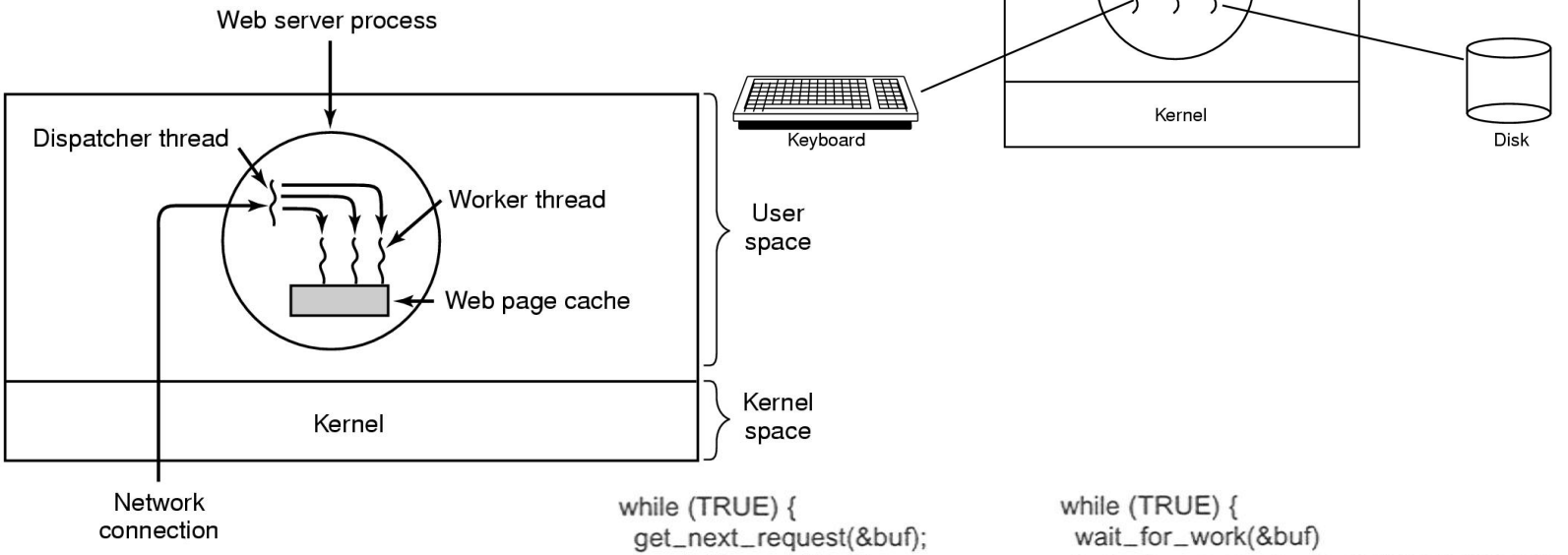
- Another useful visualization…





Why threads?

- Lower overhead than process creation
  - (usually)
- Easy to share data
- Applications scale

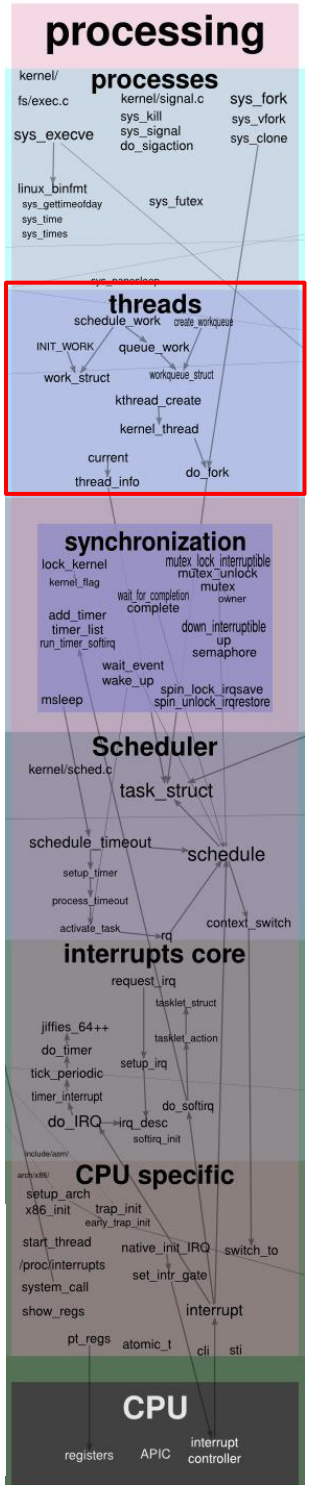



```
while (TRUE) {
  get_next_request(&buf);
  handoff_work(&buf);
}
```

(a)

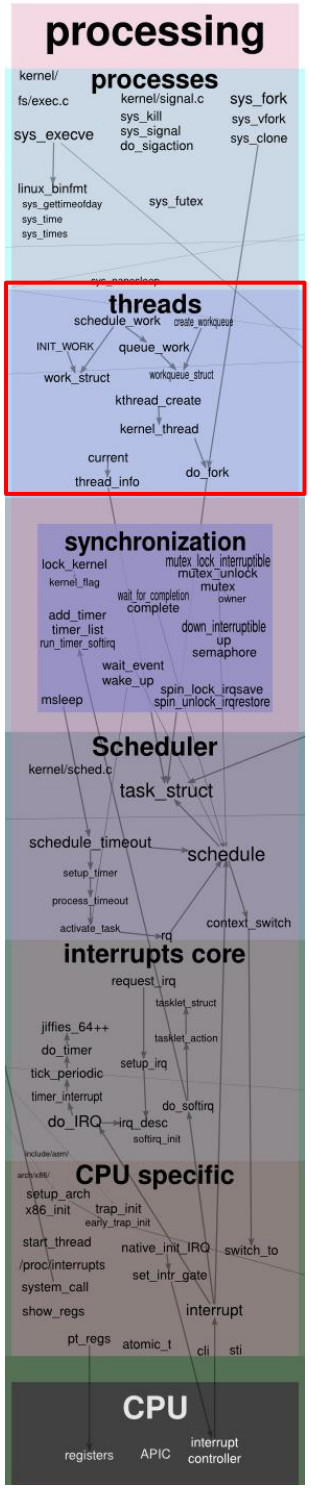
```
while (TRUE) {
  wait_for_work(&buf)
  look_for_page_in_cache(&buf, &page);
  if (page_not_in_cache(&page)
      read_page_from_disk(&buf, &page);
  return_page(&page);
}
```

(b)



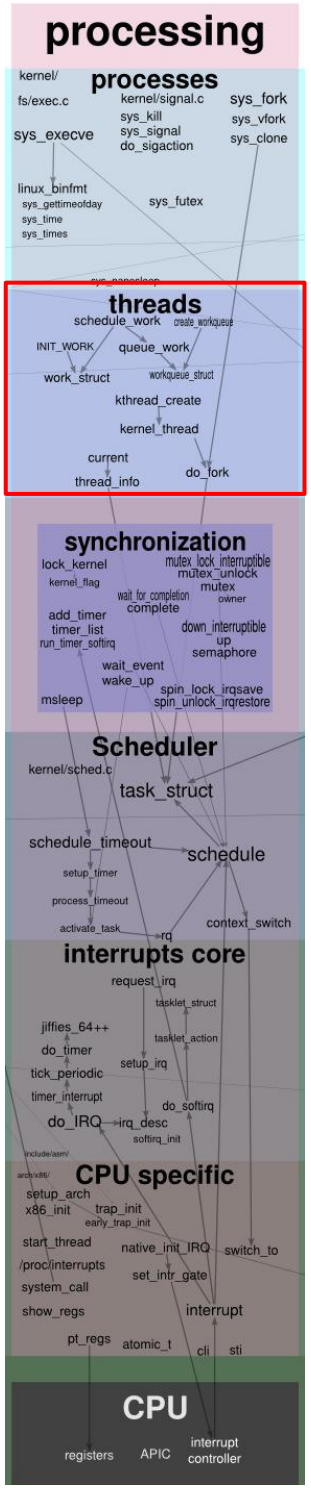
Thread benefits

- Adapt to slow devices
  - let a thread wait for I/O while another continues
- Defer work
  - do low priority stuff in background
- Exploit parallelism



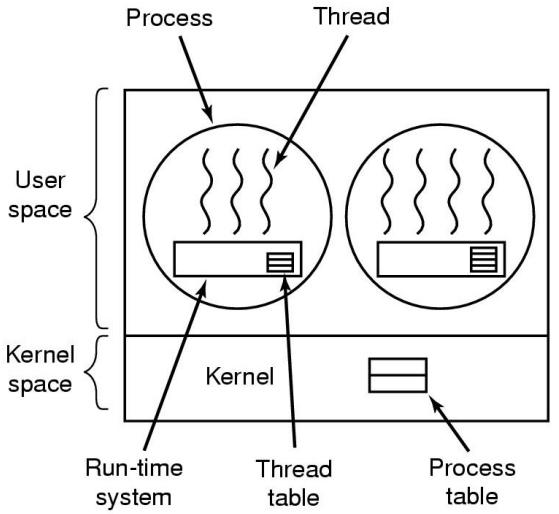
Pthread Example

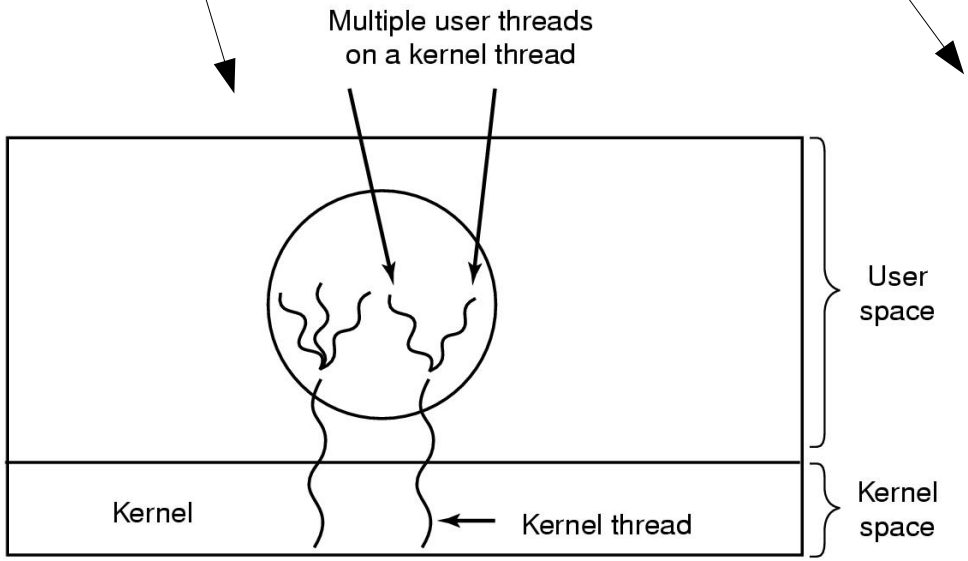
```
int main()
{
      pthread_t t1, t2;
      char *msg1 = “Thread 1”; char *msg2 = “Thread 2”;
      int ret1, ret2;
      ret1 = pthread_create(&t1, NULL, print_fn, (void*)msg1);
      ret2 = pthread_create(&t2, NULL, print_fn, (void *)msg2);
      if (ret1 || ret2) {
         fprintf(stderr, “ERROR: pthread_created failed.\n”);
         exit(1);
      }
      pthread_join(t1, NULL);
      pthread_join(t2, NULL);
      printf(“Thread 1 and thread 2 complete.\n”);
}
void print_fn(void *ptr)
{
      printf(“%s\n”, (char *)ptr);
}
```

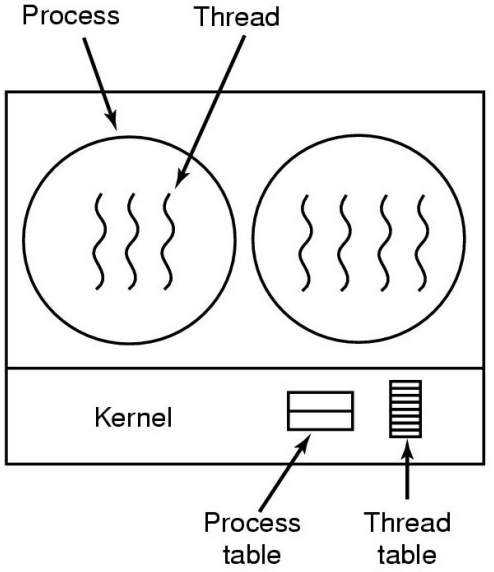


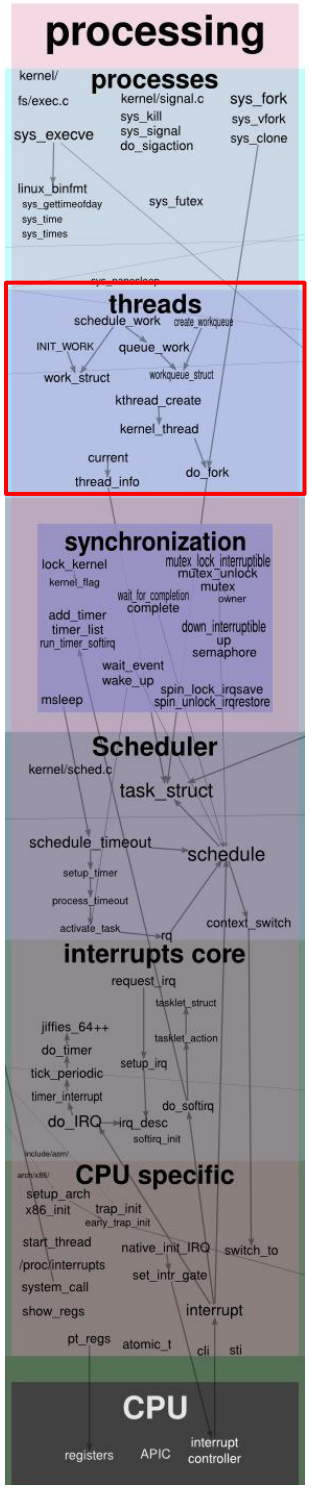
Implementation Choices

- User-level threads (N:1)
- Kernel-level threads (1:1)
- Hybrid threads (M:N)



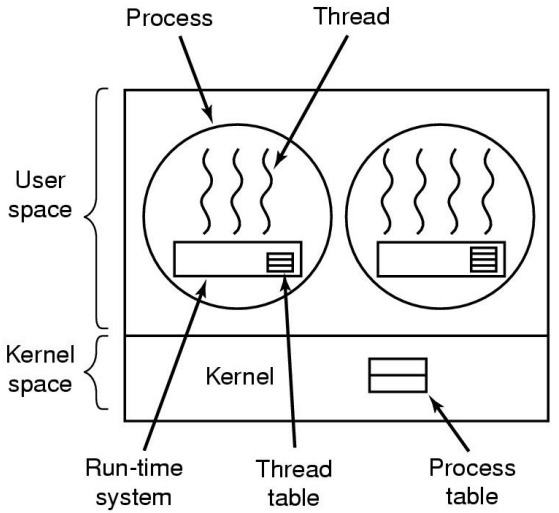


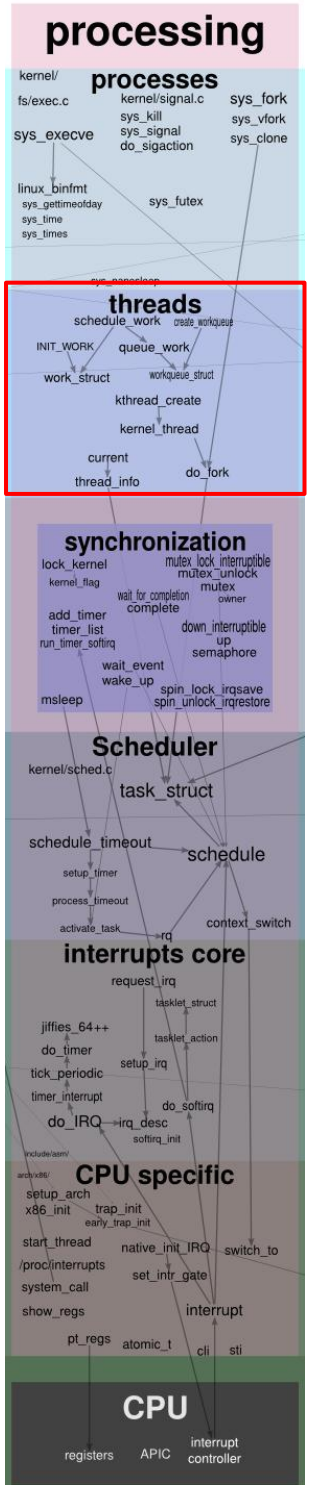




User-Level Threads (PTH, FSU…)

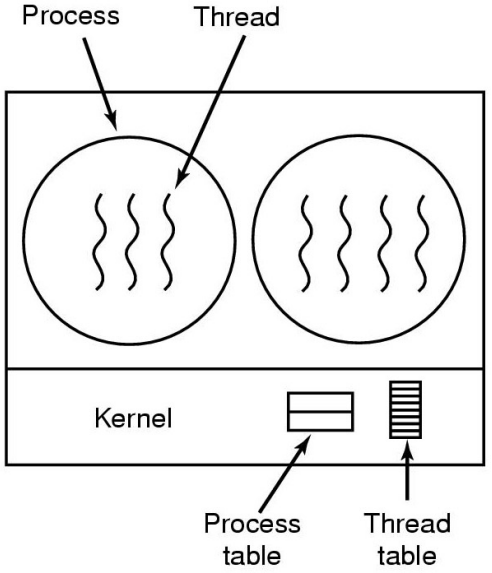
- Many-to-one mapping
- User-level libraries
  - threads created/managed at user level
  - OS not aware of threads
- Advantages:
  - No OS support required
  - Define your own scheduling policy
  - Low overhead thread ops. – no system calls
- Disadvantages:
  - No benefit on multiprocessors; all block when one blocks

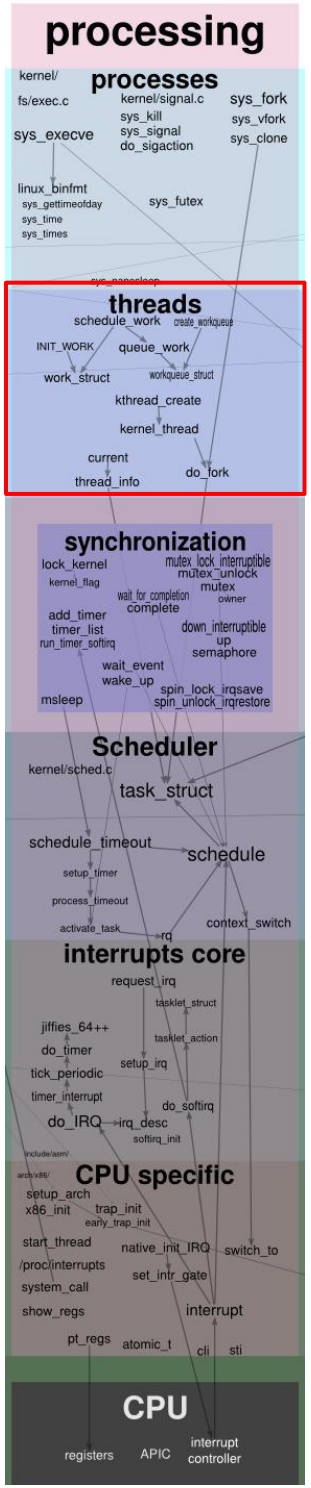




Kernel-level Threads

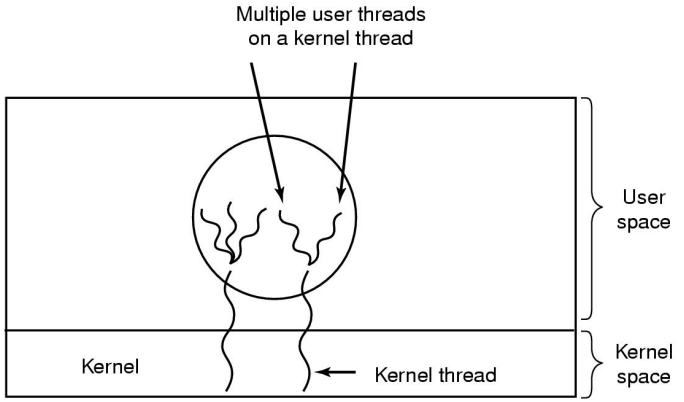
- Most common (win32, unix)
- One-to-one mapping
  - simplest design
  - one kernel task per thread
  - thread ops done by kernel
- Advantages:
  - can operate in parallel on multiprocessors
  - any thread can block without affecting others
- Disadvantages:
  - high overhead thread ops.
  - OS may not scale well with large number of threads

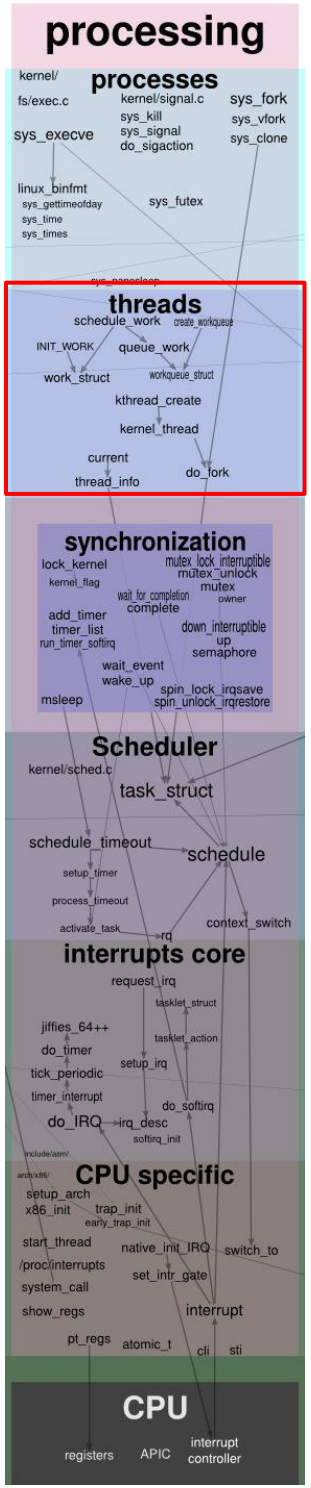




Hybrid Threads

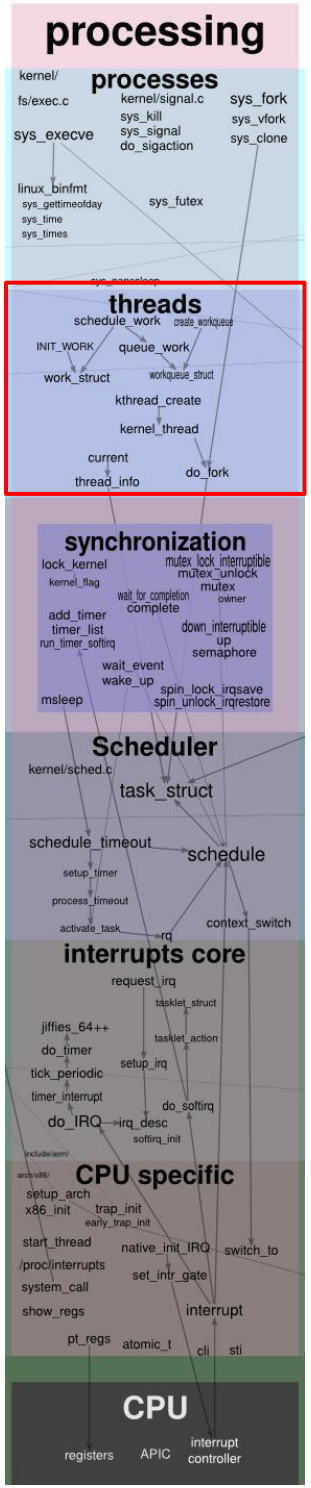
- Many-to-many mapping
- Hybrid threads (M:N)
  - App. creates m threads
  - OS creates thread pool for n kernel threads
- Advantages:
  - best of both worlds?
- Disadvantages:
  - complex
- Examples: scheduler activations





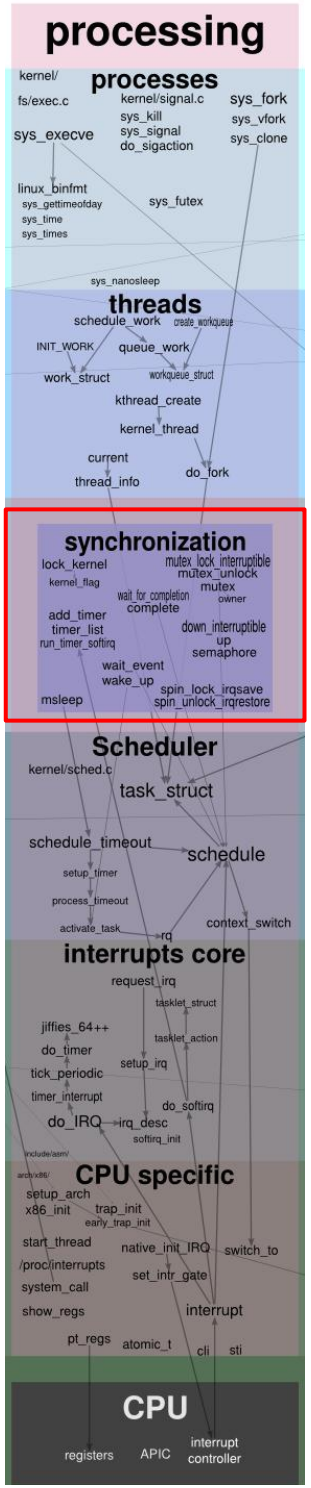
Thread Programming Models

- Manager/worker
  - a la web server, one “manager” assigns work to one or more “worker” threads
- Producer/Consumer
  - one or more producers create data/work for one or more consumers
- Pipeline
  - divide tasks into subtasks
  - handle series of subtasks by dedicated threads
- These are fundamental, but we have to understand IPC issues first, so…



Interprocess Communication (IPC)

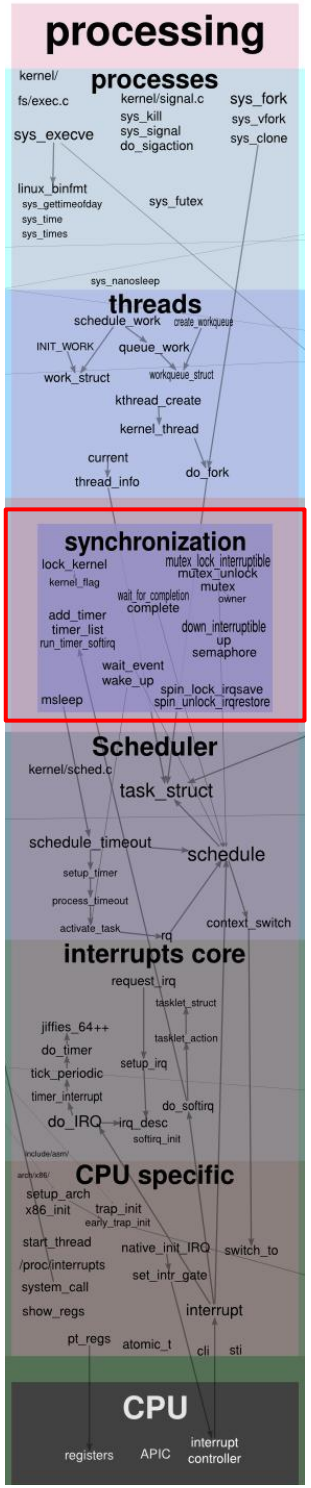
- Communication between processes is useful
- Options:
  - files, memory-mapped files, named pipes
    - process A writes to file, process B reads
  - anonymous pipes
    - A | B ^ A communicates output to B
  - socket
    - communicate through the network stack
  - shared memory
    - inherent in threads, for example
  - others: message queues, semaphores…



IPC Issues

- How do we deal with dependencies?
  - must ensure correctness
  - processes will get in each other's way
- Too much milk example:

time	You	Your Roommate
3:00	Arrive home	
3:05	Look in fridge, no milk	
3:10	Leave for grocery	
3:15		Arrive home
3:20	Arrive at grocery	Look in fridge, no milk
3:25	Buy milk	Leave for grocery
3:35	Arrive home, put milk in fridge	
3:45		Buy Milk
3:50		Arrive home, put up milk
3:50		Oh no!



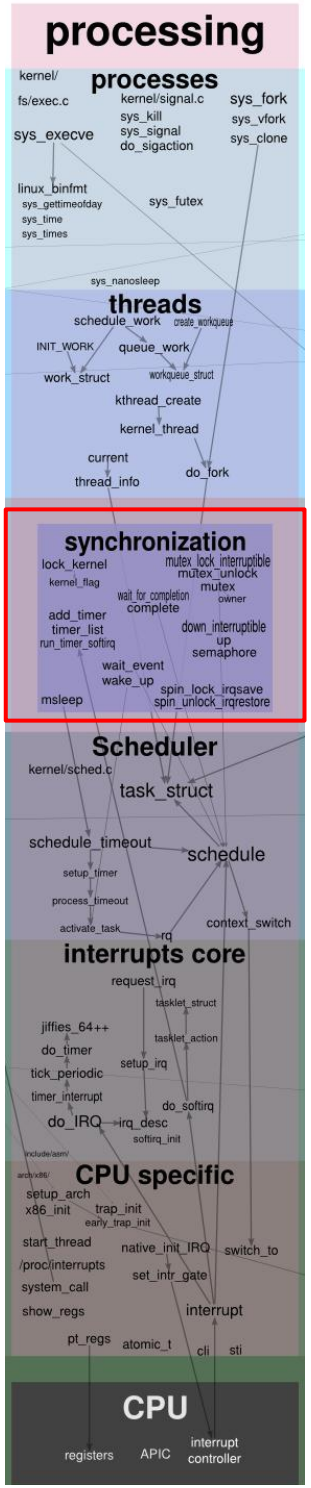
Race Conditions

- Correct operation depends on correct sequencing of events
- Notoriously difficult to debug
  - output may be nondeterministic
- Another example (a and b are in shared memory):

Thread A	Thread B
a = 0;	a = 0;
b = 0;	b = 0;
b++;	b++;
a = b;	a = b;

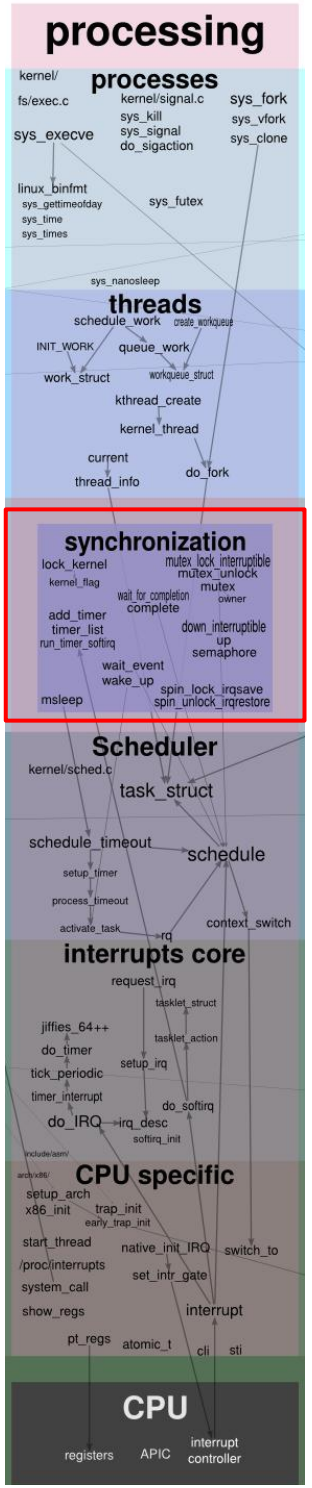
  - what are possible final values of a?

    0? 1? 2? 3? b?

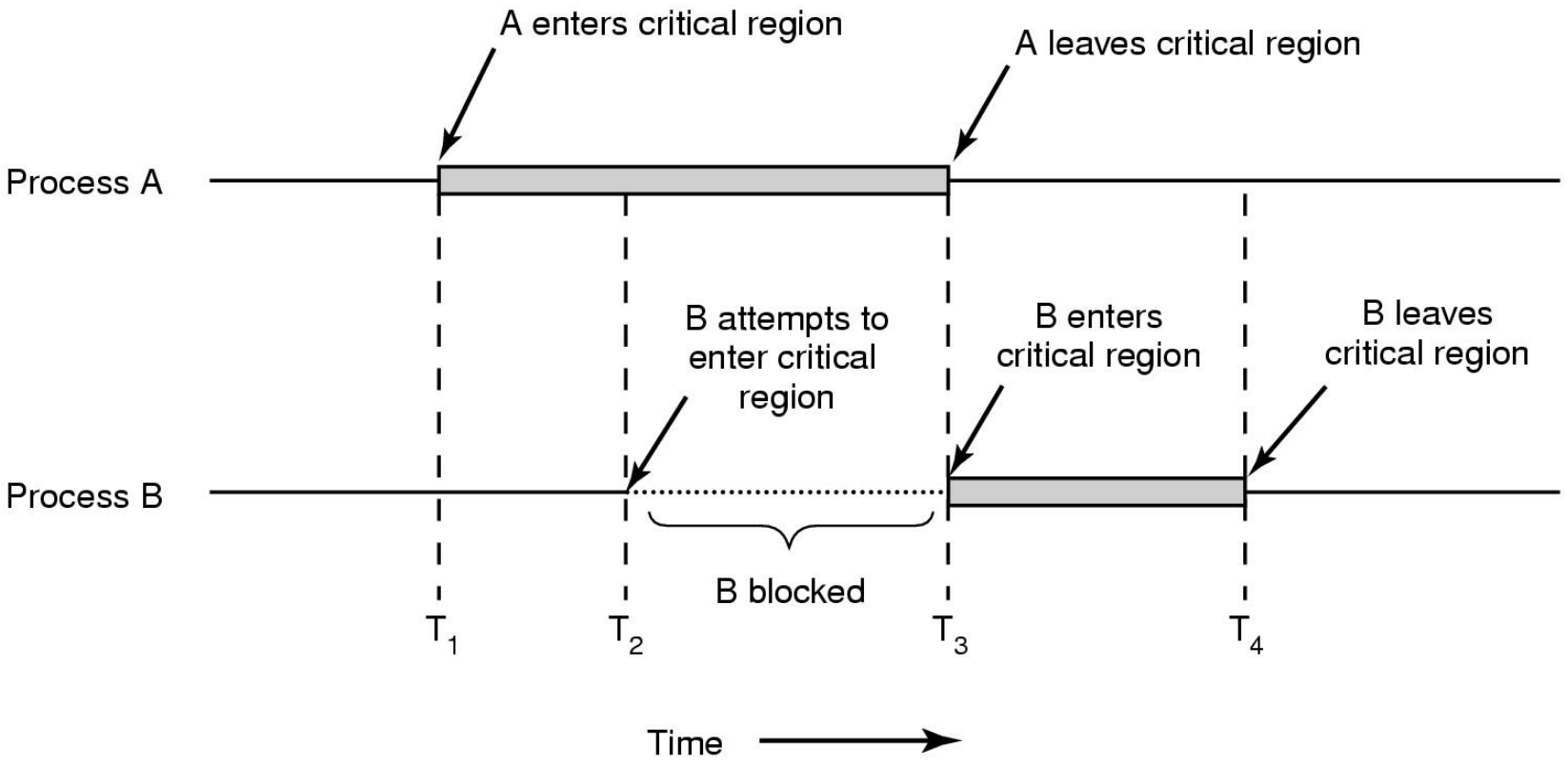


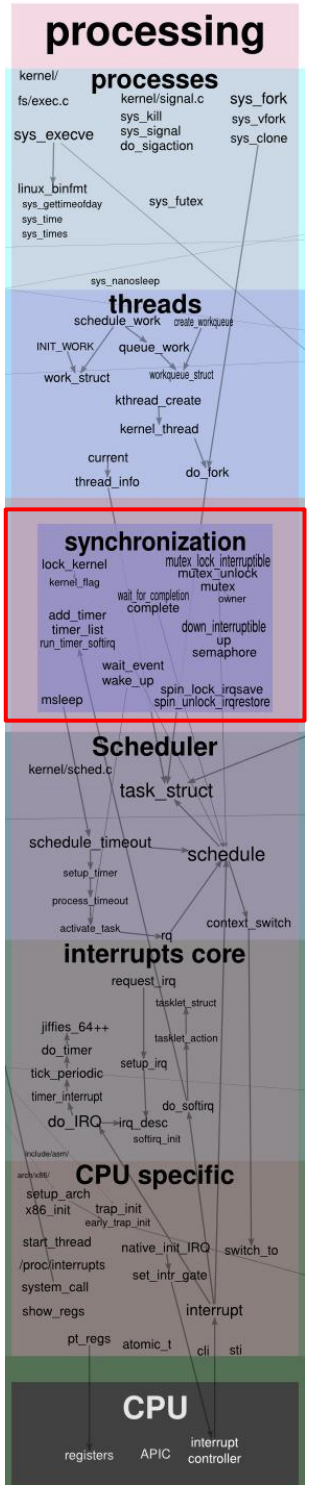
Avoiding Race Conditions

- Mutual exclusion
  - when one process is manipulating shared data, exclude other processes from accessing it
- Identify and protect Critical Sections
  - rules of the game:
    - no two processes may be in the critical section at same time
    - no assumptions may be made about number or speeds of CPUs involved
    - no process that is outside of the critical section may block other processes
    - no process should wait forever to enter the critical section



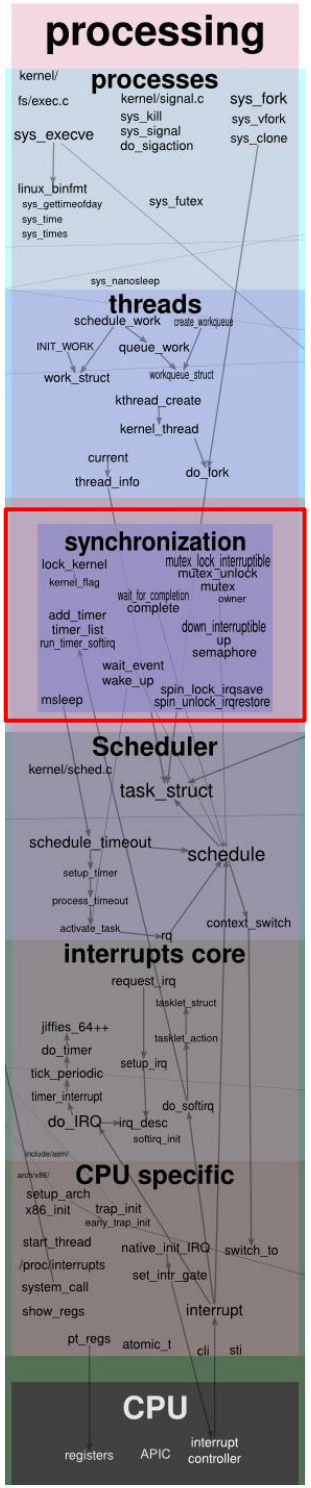
Critical Sections





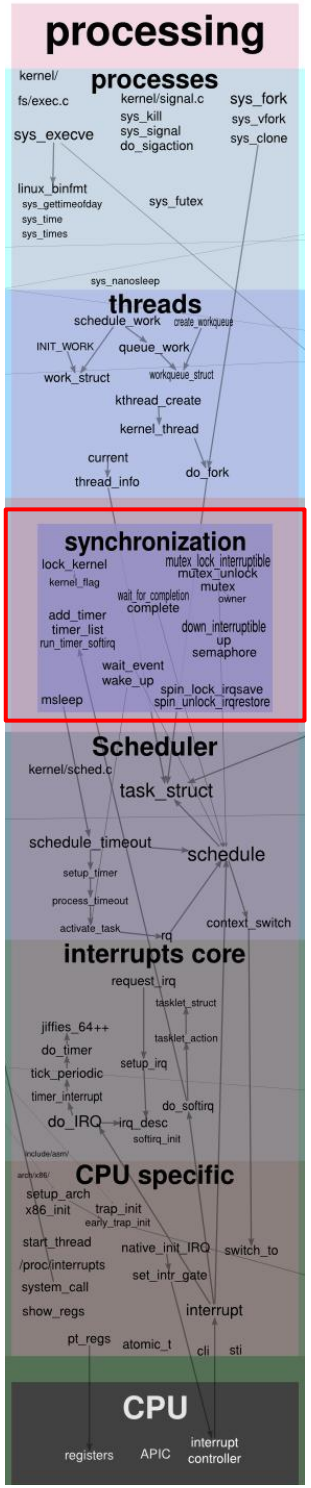
Achieving Mutual Exclusion: Disable Interrupts

- Simplest solution for uniprocessor systems
  - on multiprocessors, disabling interrupts does not provide atomicity; other CPUs can still get into critical section
- Problem:
  - process in critical section halts ^ system halts
    - allow only in OS
- DLXOS example



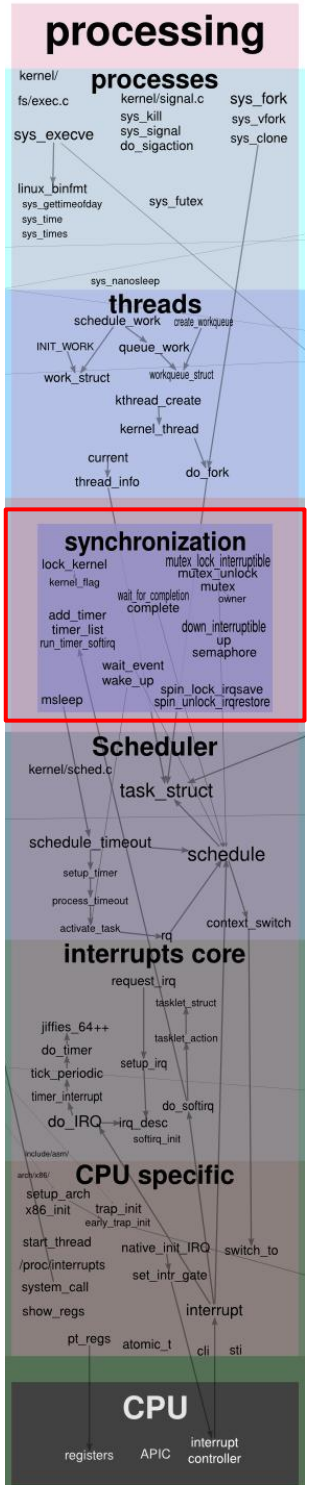
Achieving Mutual Exclusion: Read-Modify-Write

- Class of atomic operations
  - Supported by hardware
  - Examples:
    - test & set (most architectures)
      read value, write 1 to memory
    - exchange (x86)
      swap value between memory and register
    - compare & swap (68000)
      read value, if value matches register, exchange
    - load linked/conditional store (MIPS R4000, alpha)
      read value, do something, check if value modified, if not ok, else abort and jump back to start
- Busy waiting
  - waste CPU time in looping check



Achieving Mutual Exclusion: Locks

- Shared variable
  - test lock before entering critical section, if not set, set lock and enter critical section
- Can be built with HW-supported mechanisms
  - must be atomic!
  - or, could disable interrupts to make atomic
- Problems:
  - Could be problematic when threads have different priorities



Achieving Mutual Exclusion: Strict Alternation

- Simple – processes take turns

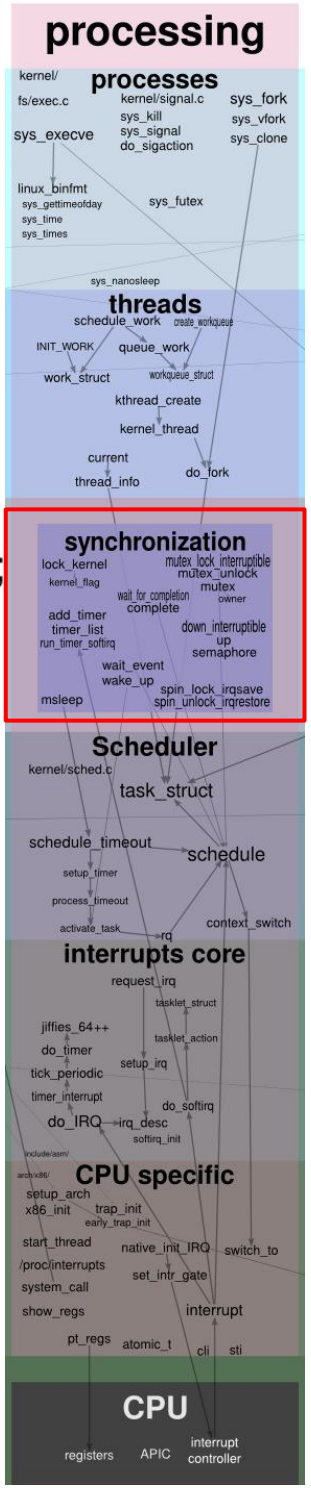
```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

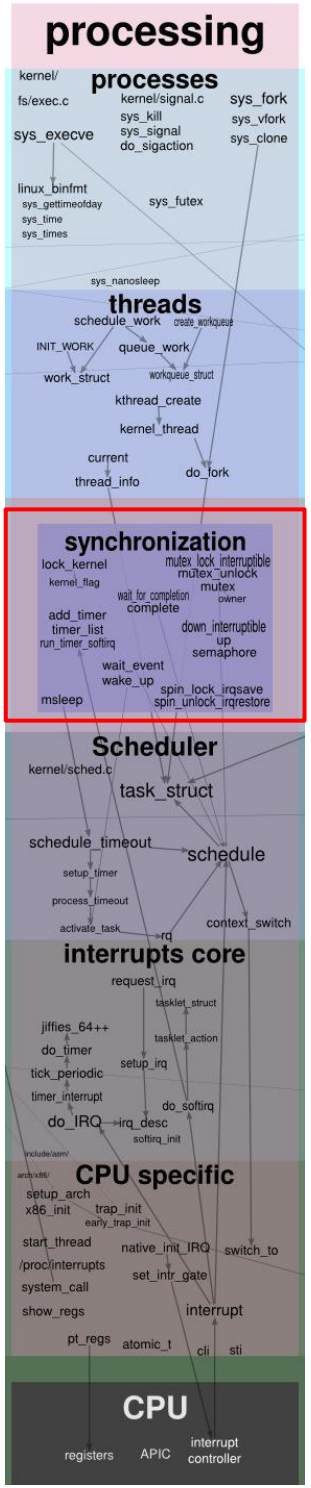
- Problem:
  - busy-waiting
  - does not scale



Achieving Mutual Exclusion: Peterson's Solution

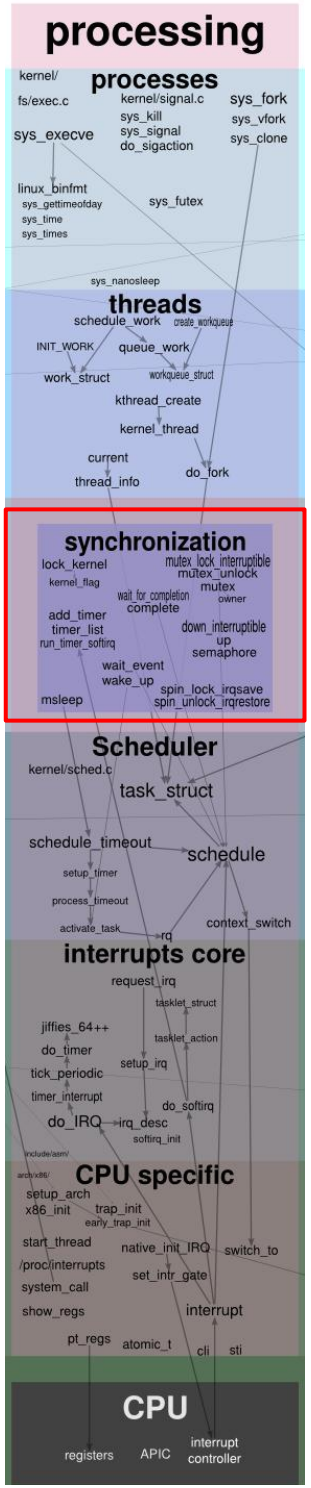
- Refinement of another solution by Dekker
- Does not require specialized HW instructions

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
      int other;                        /* number of the other process */

      other = 1 – process;              /* the opposite of process */
      interested[process] = TRUE;       /* show that you are interested */
      turn = process;                   /* set flag */
      while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
      interested[process] = FALSE;      /* indicate departure from critical region */
}
```



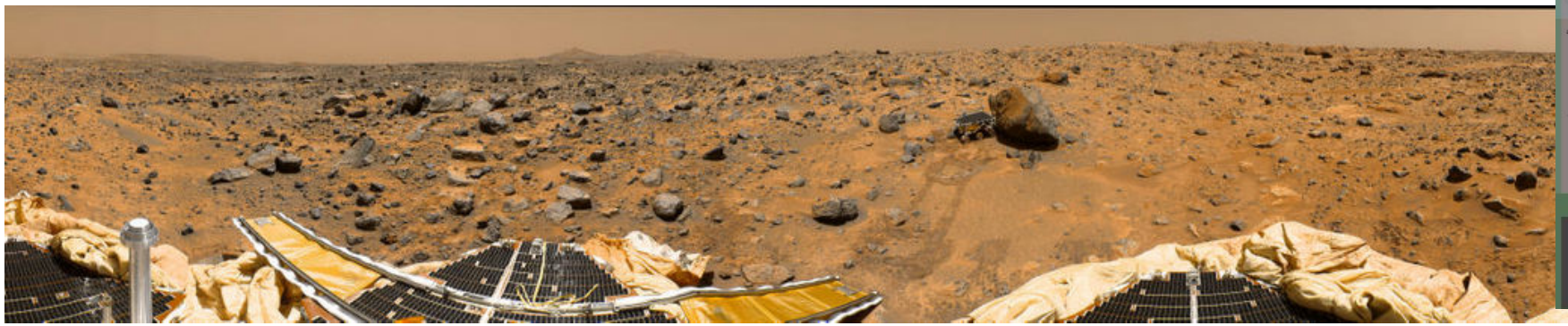
Achieving Mutual Exclusion: Peterson's Solution

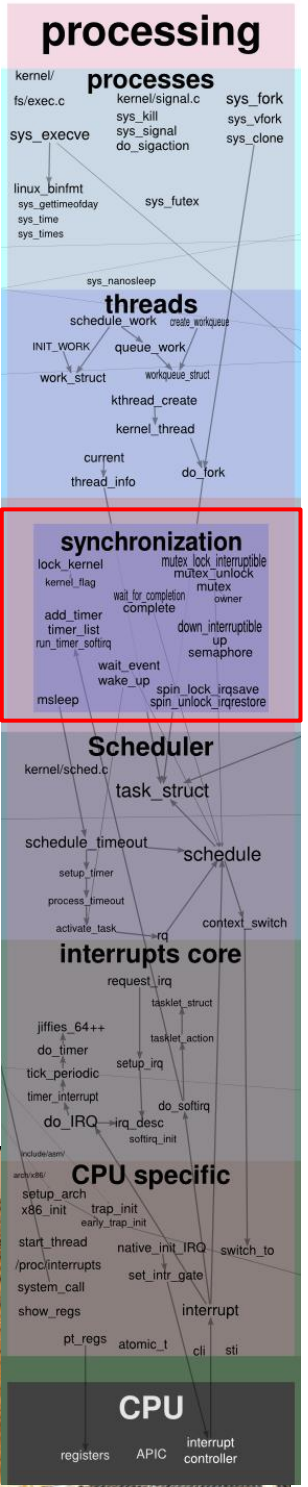
- Problems:
  - busy waiting
  - modern CPUs reorder memory access
    - requires careful implementation of Peterson's
  - became unnecessary when atomic HW operations became norm.



Priority Inversion

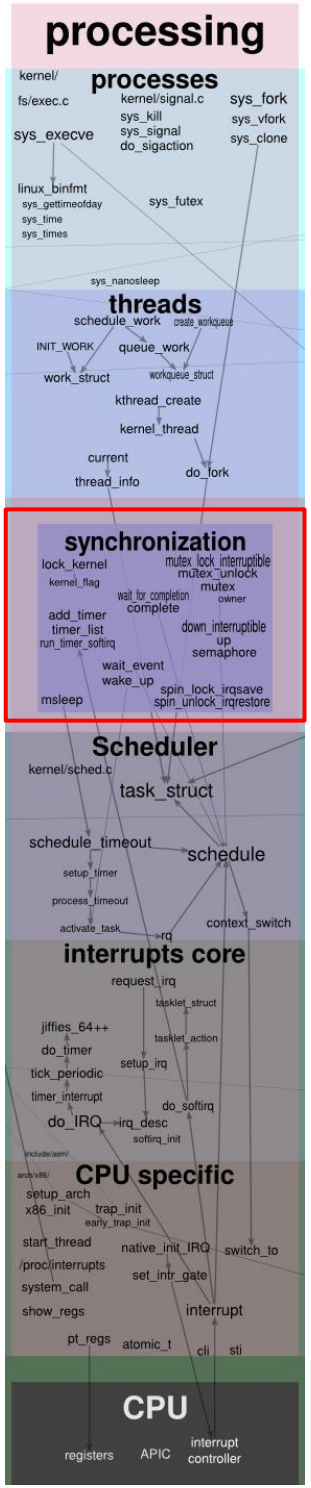
- Problem with Peterson's and read-modify-write
- Consider processes H and L
  - (High and Low priority)
  - L enters critical region
  - H preempts L
  - L swapped out while in critical region
  - H begins busy waiting, since L never exited cs
    - it never stops





Don't Wait, Block

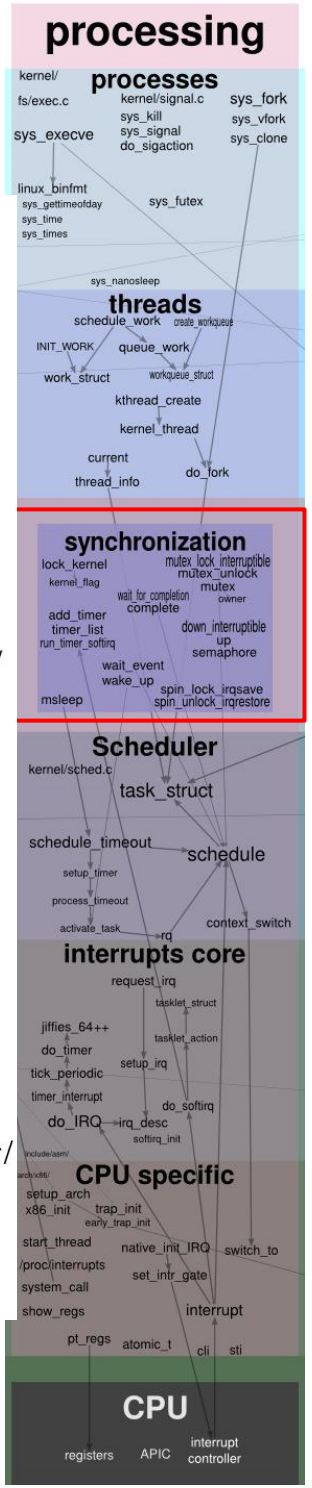
- Replacing busy-wait with sleep and wake
  - good idea for multiprocessors
- Sleep
  - put process to sleep when resource unavailable
  - no wasted cycles
- Wake
  - resource producer signals sleeping thread
- Need queue to implement



Sleep-Wakeup: Producer/Consumer

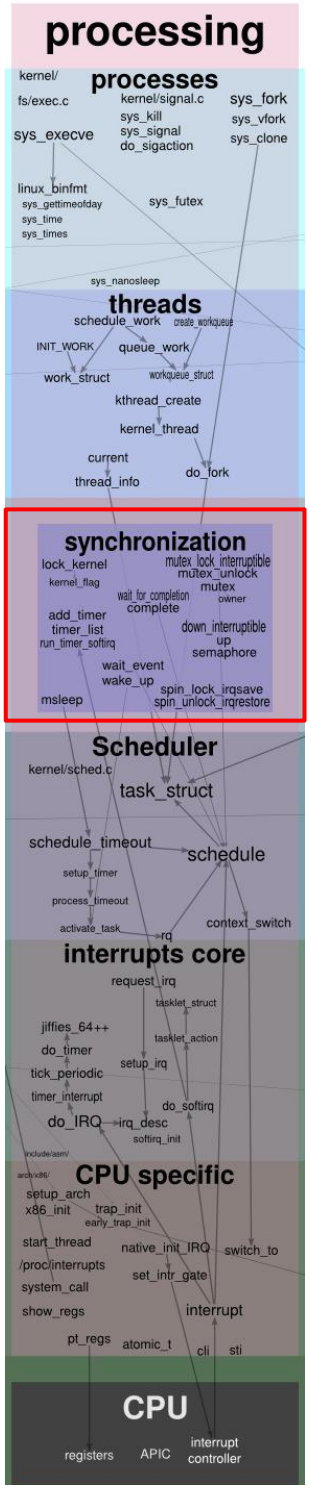
- Lost wakeup
- Access to count ^ race cond.
- How?

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}

void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count – 1;                    /* decrement count of items in buffer */
          if (count == N – 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```



Semaphores

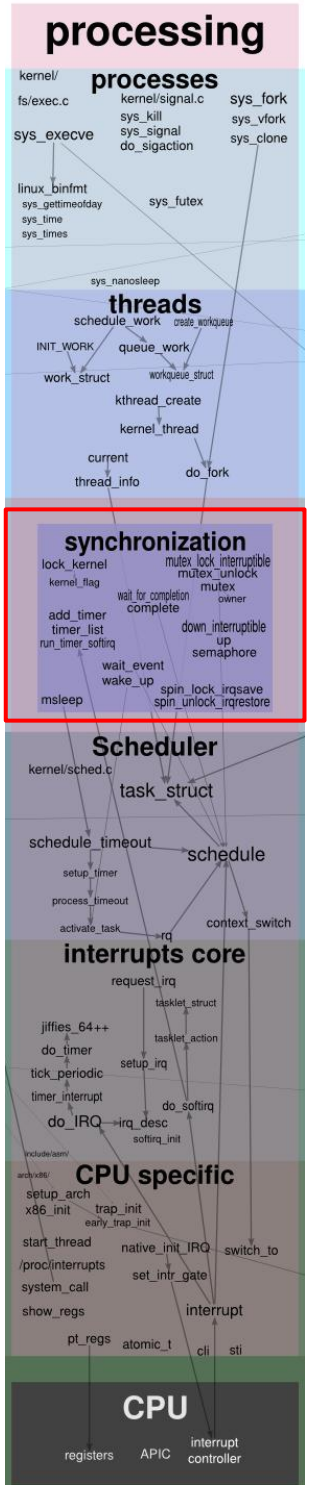
- General solution
- Need queue and two operations:
  - signal/post/up/V: increments semaphore by 1
    wakes up a waiting process on queue
    (atomic)
  - wait/down/P: decrement semaphore by 1
    puts process to sleep if semaphore exhausted
    (atomic)



Semaphores (Implementation)

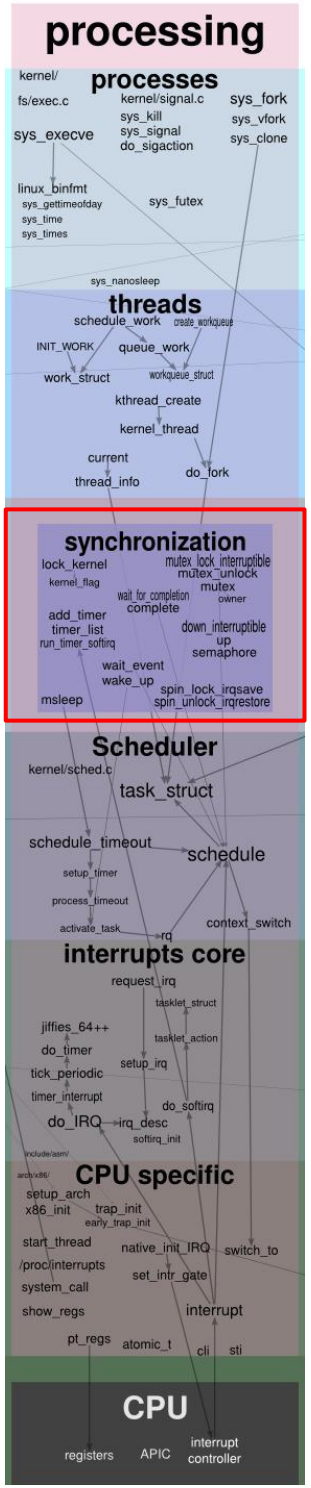
```
void SemWait(Sem* sem)
{
    //...
    intrs = DisableIntrs();
    sem->count -= 1;
    if (sem->count < 0) {
    //...
        QueueInsertLast (&sem->waiting, l);
        ProcessSleep ();
    }
    RestoreIntrs(intrs);
}

void SemSignal(Sem *sem)
{
    //...
    intrs = DisableIntrs();
    sem->count += 1;
    if (sem->count <= 0) {
        //...
        ProcessWakeup ((PCB *)(l->object));
        //...
    }
    RestoreIntrs(intrs);
}
```

```
typedef struct Sem
{
    Queue waiting;
    int count;
} Sem;
```



Uses for Semaphores

- Counting semaphore
  - use protected count value as counter
- Mutex
  - binary semaphore (initialized to 1)
  - used to limit access to a critical section
- Resource management
  - initialize semaphore to number resource instances



Basic synchronization: signaling

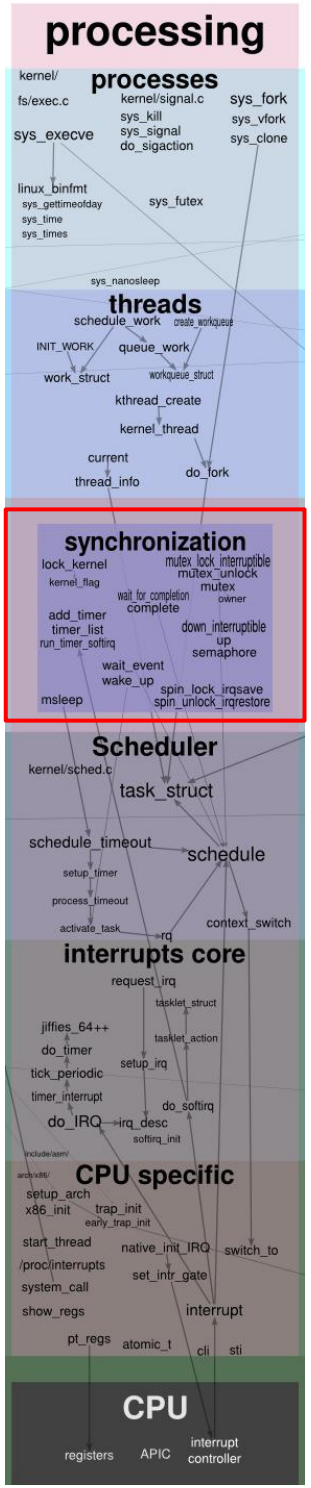
- How do you ensure that a section of code in one thread runs before a section of code in another?

Thread A

```
1  statement a1
2  sem.signal()
```

Thread B

```
1  sem.wait()
2  statement b1
```



Basic synchronization: rendezvous

Thread A

```
1  statement a1
2  statement a2
```

Thread B

```
1  statement b1
2  statement b2
```

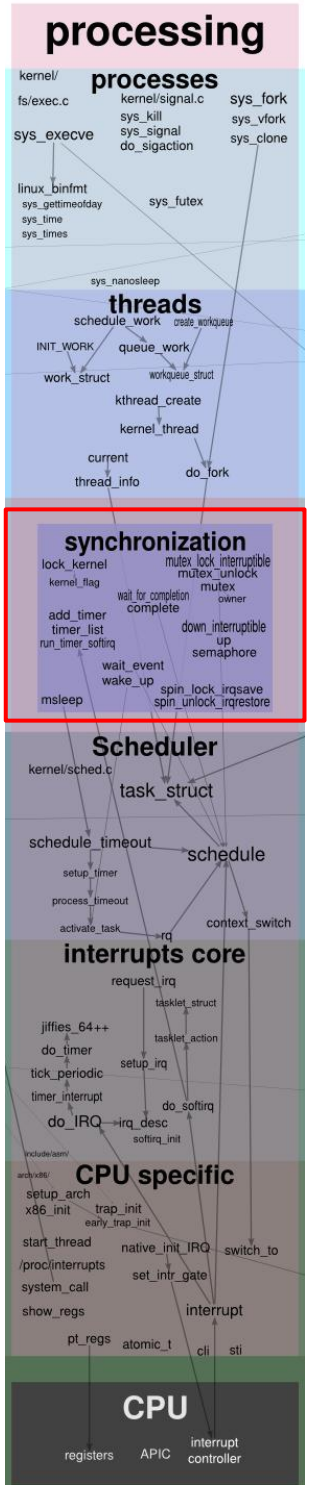
- How do you guarantee that a1 happens before b2 and b1 happens before a2?

Thread A

```
1  statement a1
2  aArrived.signal()
3  bArrived.wait()
4  statement a2
```

Thread B

```
1  statement b1
2  bArrived.signal()
3  aArrived.wait()
4  statement b2
```



Basic synchronization: mutex

Thread A

```
count = count + 1
```

Thread B

```
count = count + 1
```

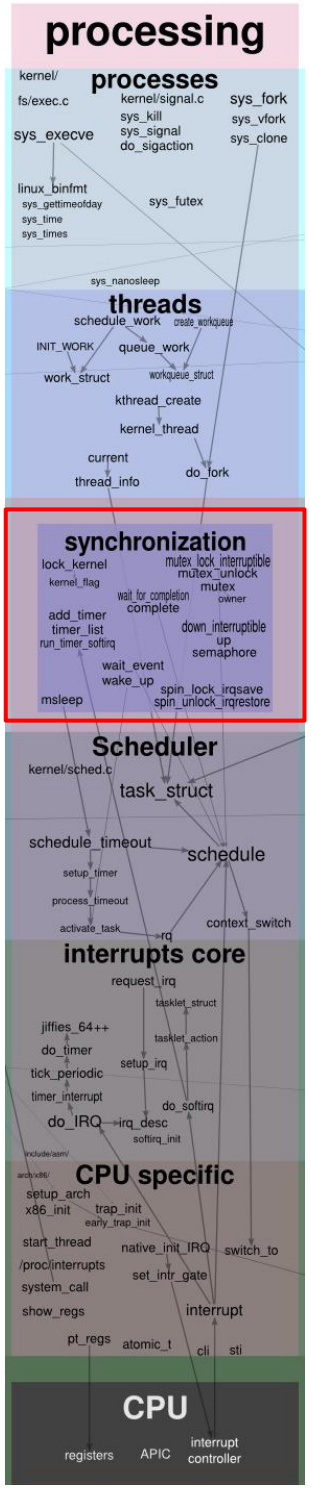
- How do you control access to `count`?

Thread A

```
mutex.wait()
    # critical section
    count = count + 1
mutex.signal()
```

Thread B

```
mutex.wait()
    # critical section
    count = count + 1
mutex.signal()
```

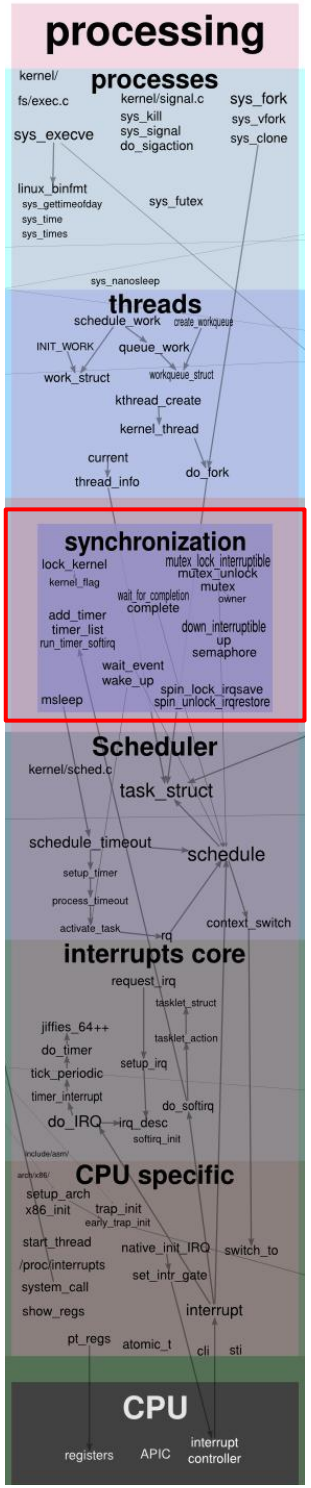


Basic synchronization: multiplex

- You may find it useful to allow multiple threads into the critical section

```
1  multiplex.wait()
2      critical section
3  multiplex.signal()
```

- Here, multiplex is a semaphore initialized to n
  - where n is the number of threads allowed to enter



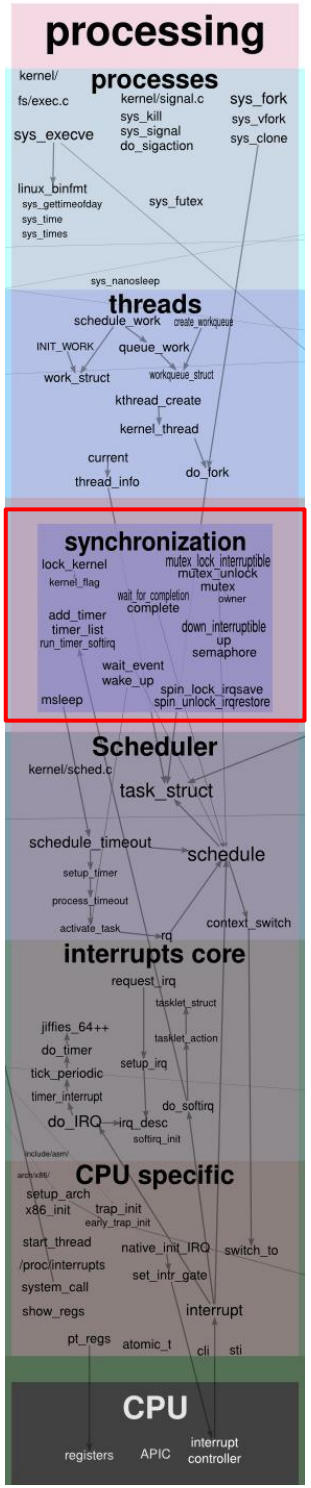
Basic synchronization: barrier

```
1  rendezvous
2  critical point
```

- Generalized rendezvous
  - rendezvous only works for 2 threads

```
1  rendezvous
2
3  mutex.wait()
4      count = count + 1
5  mutex.signal()
6
7  if count == n: barrier.signal()
8
9  barrier.wait()
10 barrier.signal()
11
12 critical point
```

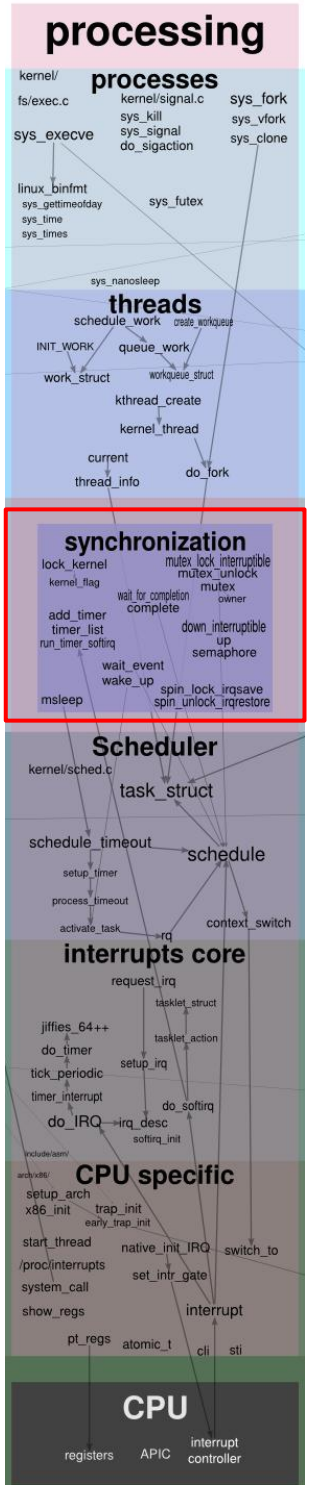
turnstile



Basic synchronization: reusable barrier

- Also known as two-phase barrier
- How do you ensure that threads synchronize at each iteration of a loop?
- Barrier solution below is insufficient, why?

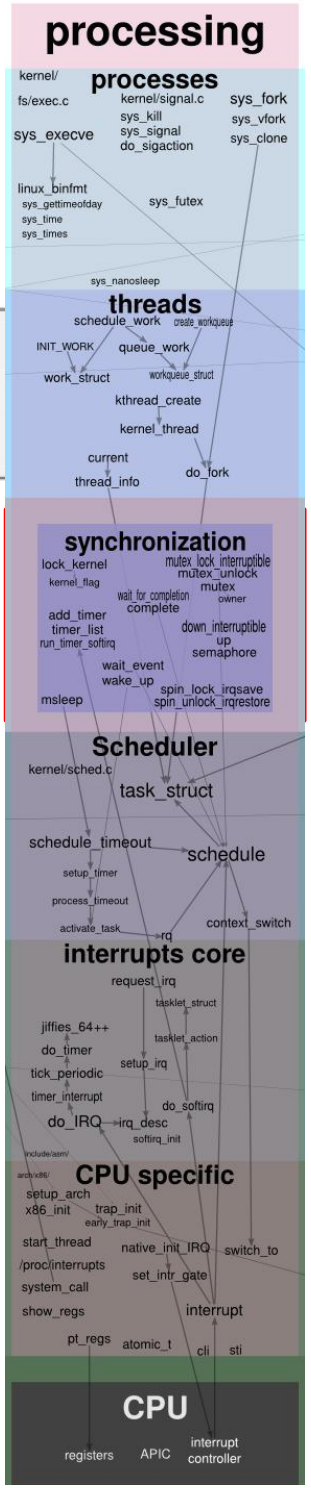
```
1  rendezvous
2
3  mutex.wait()
4      count = count + 1
5  mutex.signal()
6
7  if count == n: barrier.signal()
8
9  barrier.wait()
10 barrier.signal()
11
12 critical point
```



Basic synchronization: reusable barrier

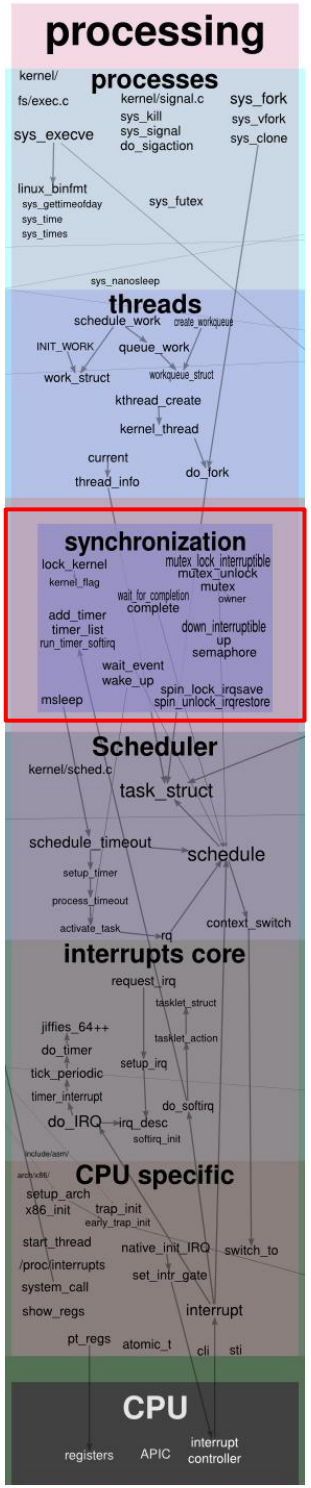
```
1  turnstile = Semaphore(0)
2  turnstile2 = Semaphore(1)
3  mutex = Semaphore(1)
```

```
1  # rendezvous
2
3  mutex.wait()
4      count += 1
5      if count == n:
6          turnstile2.wait()     # lock the second
7          turnstile.signal()    # unlock the first
8  mutex.signal()
9
10 turnstile.wait()              # first turnstile
11 turnstile.signal()
12
13 # critical point
14
15 mutex.wait()
16     count -= 1
17     if count == 0:
18         turnstile.wait()      # lock the first
19         turnstile2.signal()   # unlock the second
20 mutex.signal()
21
22 turnstile2.wait()            # second turnstile
23 turnstile2.signal()
```



Producer/Consumer with Semaphores

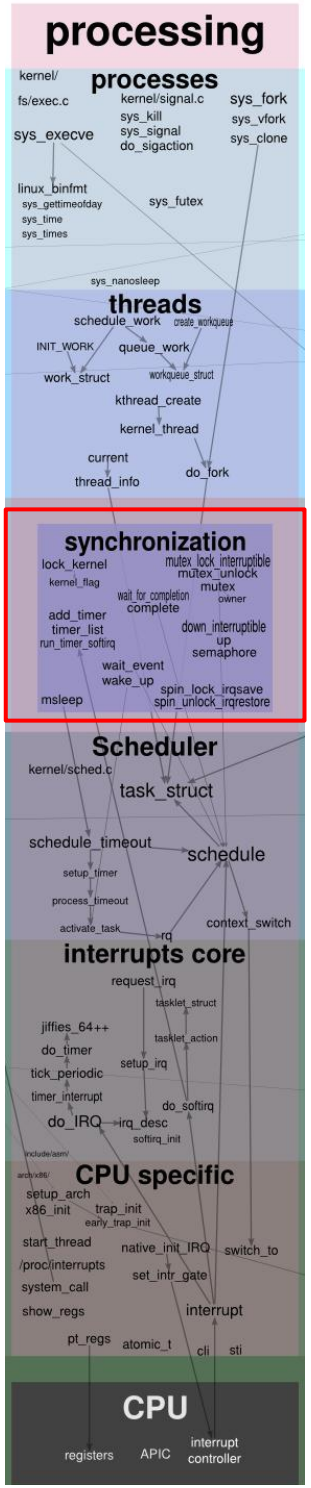
```
#define N 100                       /* number of slots in the buffer */
typedef int semaphore;              /* semaphores are a special kind of int */
semaphore mutex = 1;                /* controls access to critical region */
semaphore empty = N;                /* counts empty buffer slots */
semaphore full = 0;                 /* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {                  /* TRUE is the constant 1 */
        item = produce_item( );     /* generate something to put in buffer */
        down(&empty);               /* decrement empty count */
        down(&mutex);               /* enter critical region */
        insert_item(item);          /* put new item in buffer */
        up(&mutex);                 /* leave critical region */
        up(&full);                  /* increment count of full slots */
    }
}


void consumer(void)
{
    int item;

    while (TRUE) {                  /* infinite loop */
        down(&full);                /* decrement full count */
        down(&mutex);               /* enter critical region */
        item = remove_item( );      /* take item from buffer */
        up(&mutex);                 /* leave critical region */
        up(&empty);                 /* increment count of empty slots */
        consume_item(item);         /* do something with the item */
    }
}
```



Danger of Semaphores

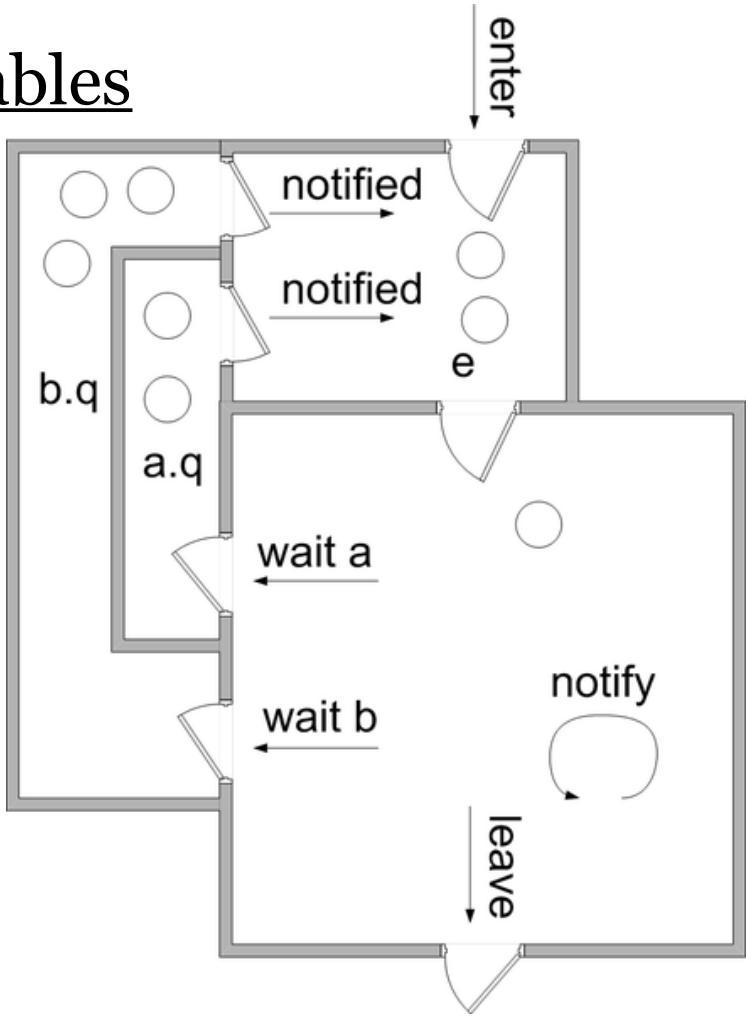
- What happens in this situation?

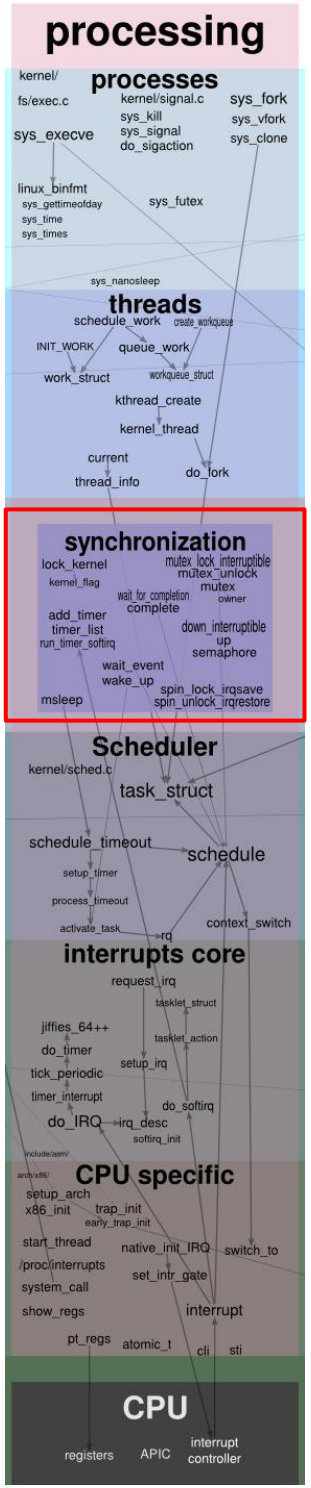
task 1	task 2
wait(semA);	wait(semB);
wait(semB);	wait(semA);
//c.s...	//c.s...
signal(semA);	signal(semA);
signal(semB);	signal(semB);



Monitors

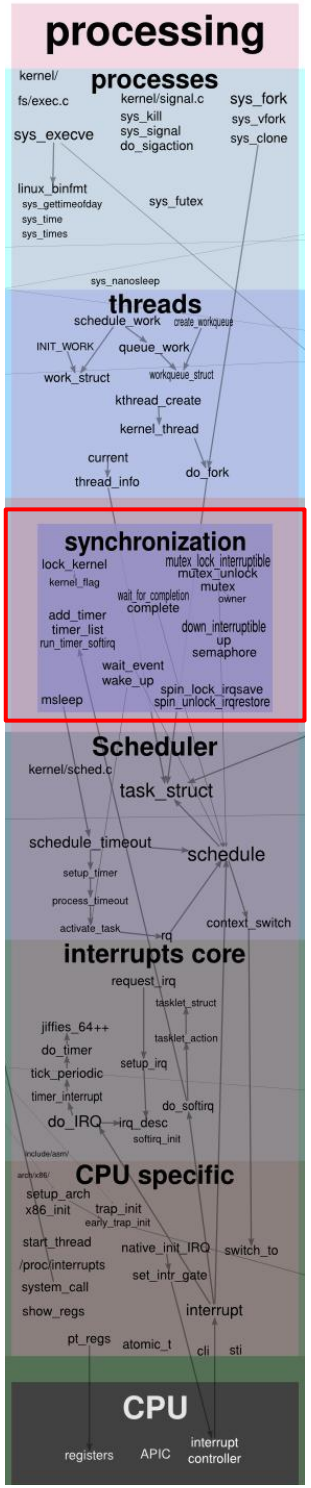
- a and b are condition variables
- Primitives:
  - wait: places thread on queue
  - notify: notifies condition variable that condition has been met (frees a thread from a queue)





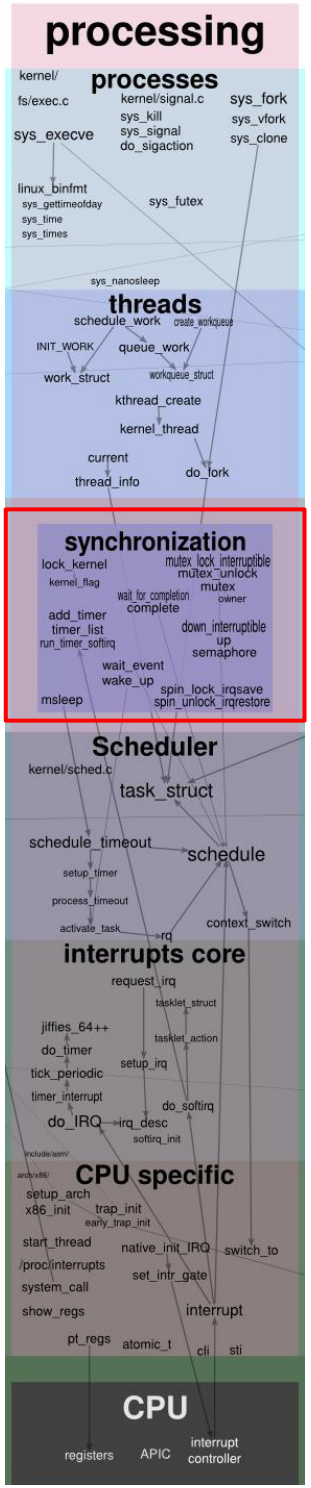
Distributed systems solutions

- Thus far, we've seen only shared-memory synchronization solutions
  - processes have access to same memory
  - may work in distributed systems, but memory management gets complex
- Distributed memory solutions?
  - cannot depend on shared variables
  - cannot necessarily access same address space
  - based on message passing



Message Passing (distributed systems)

- Primitives:
  - Send
    - blocking/rendezvous: sender waits for receiver
    - buffered non-blocking: OS buffers, sender moves on
    - non-buffered non-blocking: error if receiver not ready or waiting
  - Receive
    - blocking/rendezvous: wait for message
    - buffered non-blocking: return error if no message waiting
    - non-buffered non-blocking: error if no sender ready



IPC Classics

- Dining philosophers
- Readers and writers
- Sleeping barber

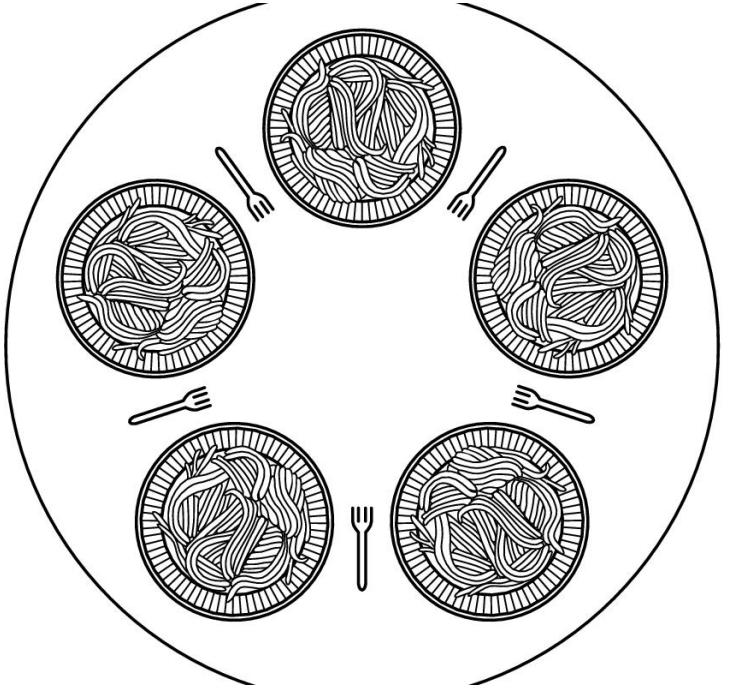
Dining Philosophers

- Philosophers think/eat
- Eating requires 2 forks
- Can only pick 1 fork up

at a time

- Must prevent starvation

and deadlock



- Models multiprocess competition for limited resources

Naïve Solution

```
#define N 5                                     /* number of philosophers */

void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */
{
        while (TRUE) {
                think( );                       /* philosopher is thinking */
                take_fork(i);                   /* take left fork */
                take_fork((i+1) % N);           /* take right fork; % is modulo operator */
                eat( );                         /* yum-yum, spaghetti */
                put_fork(i);                    /* put left fork back on the table */
                put_fork((i+1) % N);            /* put right fork back on the table */
        }
}
```

Why doesn't this work?

Solution

```
#define N                5                    /* number of philosophers */
#define LEFT             (i+N–1)%N            /* number of i's left neighbor */
#define RIGHT            (i+1)%N              /* number of i's right neighbor */
#define THINKING         0                    /* philosopher is thinking */
#define HUNGRY           1                    /* philosopher is trying to get forks
#define EATING           2                    /* philosopher is eating */
typedef int semaphore;                        /* semaphores are a special kind of
int state[N];                                 /* array to keep track of everyone's
semaphore mutex = 1;                          /* mutual exclusion for critical regio
semaphore s[N];                               /* one semaphore per philosopher

void philosopher(int i)                       /* i: philosopher number, from 0 to
{
        while (TRUE) {                        /* repeat forever */
                think( );                     /* philosopher is thinking */
                take_forks(i);                /* acquire two forks or block */
                eat( );                       /* yum-yum, spaghetti */
                put_forks(i);                 /* put both forks back on table */
        }
}
```

Solution(2)

```
void take_forks(int i)                  /* i: philosopher number, from 0 to N−1 */
{
        down(&mutex);                   /* enter critical region */
        state[i] = HUNGRY;              /* record fact that philosopher i is hungry */
        test(i);                        /* try to acquire 2 forks */
        up(&mutex);                     /* exit critical region */
        down(&s[i]);                    /* block if forks were not acquired */
}

void put_forks(i)                       /* i: philosopher number, from 0 to N−1 */
{
        down(&mutex);                   /* enter critical region */
        state[i] = THINKING;            /* philosopher has finished eating */
        test(LEFT);                     /* see if left neighbor can now eat */
        test(RIGHT);                    /* see if right neighbor can now eat */
        up(&mutex);                     /* exit critical region */
}

void test(i) /* i: philosopher number, from 0 to N−1 */
{
        if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
                state[i] = EATING;
                up(&s[i]);
        }
}
```

Readers and Writers

- Models access to a database
- Multiple processes reading
- Can have only one writer
  - When writer is present, no others have access

Solution

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to 'rc' */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
        while (TRUE) {                  /* repeat forever */
                down(&mutex);           /* get exclusive access to 'rc' */
                rc = rc + 1;            /* one reader more now */
                if (rc == 1) down(&db); /* if this is the first reader ... */
                up(&mutex);             /* release exclusive access to 'rc' */
                read_data_base( );      /* access the data */
                down(&mutex);           /* get exclusive access to 'rc' */
                rc = rc – 1;            /* one reader fewer now */
                if (rc == 0) up(&db);   /* if this is the last reader ... */
                up(&mutex);             /* release exclusive access to 'rc' */
                use_data_read( );       /* noncritical region */
        }
}
```

Solution(2)

```
void writer(void)
{
        while (TRUE) {                          /* repeat forever */
                think_up_data( );               /* noncritical region */
                down(&db);                      /* get exclusive access */
                write_data_base( );             /* update the data */
                up(&db);                        /* release exclusive access */
        }
}
```

Sleeping Barber

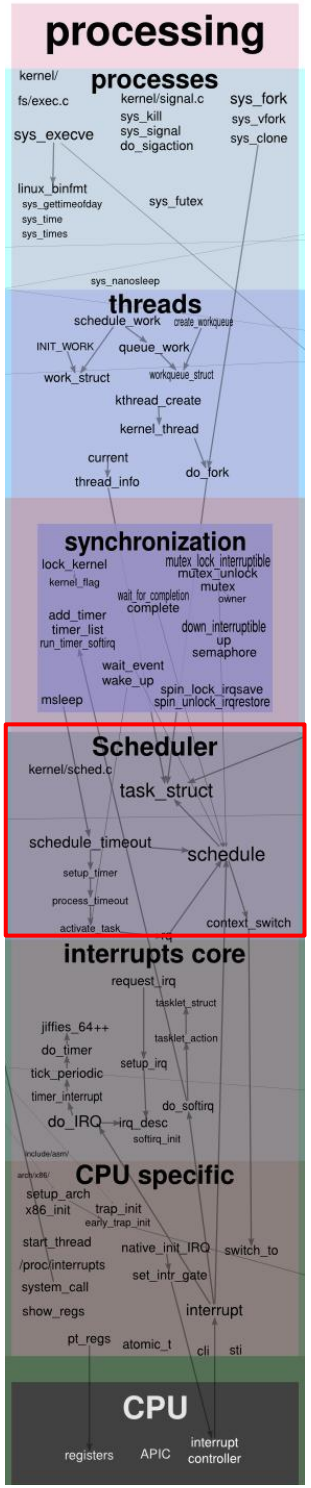
- Barber shop
  - one barber
  - one barber chair
  - several waiting chairs
- New customer
  - if barber is sleeping, wake him up
  - if barber is busy, go to waiting chair
- Barber
  - when finished, dismiss customer, check waiting chairs
    - if nobody waiting, go to sleep
- Problem?

Solution

```
#define CHAIRS 5                    /* # chairs for waiting customers */

typedef int semaphore;              /* use your imagination */

semaphore customers = 0;            /* # of customers waiting for service */
semaphore barbers = 0;              /* # of barbers waiting for customers */
semaphore mutex = 1;                /* for mutual exclusion */
int waiting = 0;                    /* customers are waiting (not being cut) */

void barber(void)
{
    while (TRUE) {
        down(&customers);           /* go to sleep if # of customers is 0 */
        down(&mutex);               /* acquire access to 'waiting' */
        waiting = waiting – 1;      /* decrement count of waiting customers */
        up(&barbers);               /* one barber is now ready to cut hair */
        up(&mutex);                 /* release 'waiting' */
        cut_hair( );                /* cut hair (outside critical region) */
    }
}


void customer(void)
{
    down(&mutex);                   /* enter critical region */
    if (waiting < CHAIRS) {         /* if there are no free chairs, leave */
        waiting = waiting + 1;      /* increment count of waiting customers */
        up(&customers);             /* wake up barber if necessary */
        up(&mutex);                 /* release access to 'waiting' */
        down(&barbers);             /* go to sleep if # of free barbers is 0 */
        get_haircut( );             /* be seated and be serviced */
    } else {
        up(&mutex);                 /* shop is full; do not wait */
    }
}
```

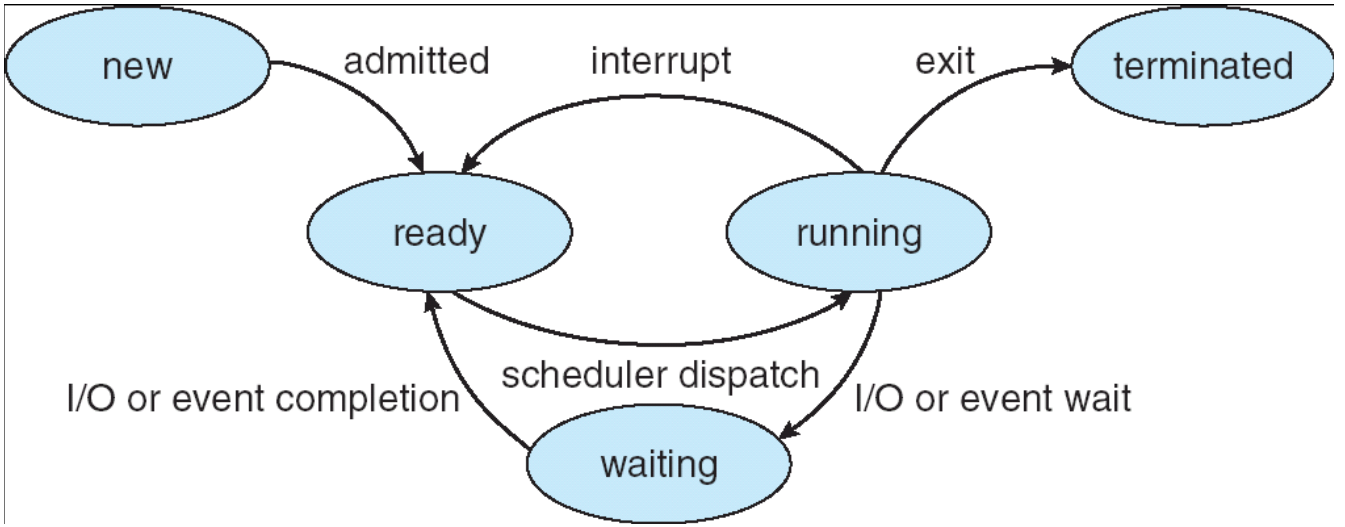
Question

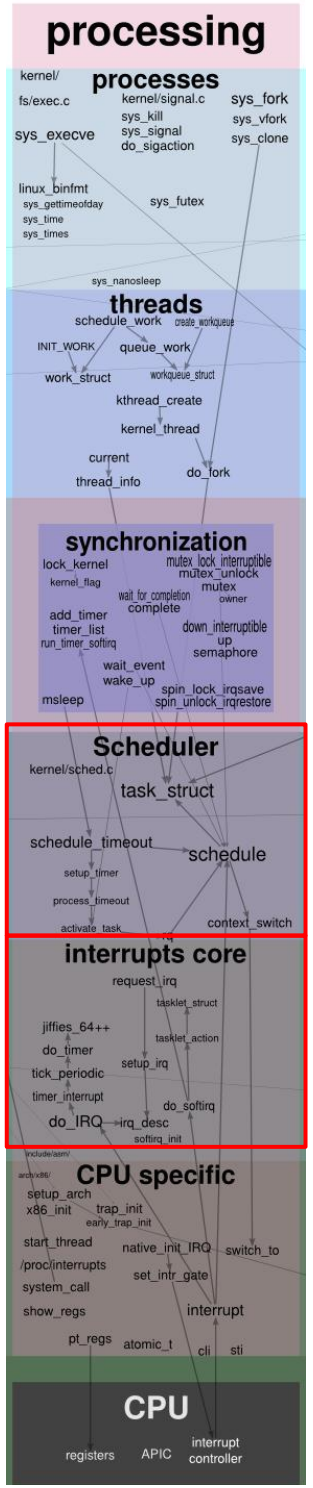
- To achieve multi-tasking/multiprogramming is it sufficient to rely on processes being cooperative?
- As in, do we allow processes to dictate when the scheduler is invoked and next process given an opportunity to run? Is this a good idea?
- Probably not:
  - processes misbehave: avoiding traps and I/O can lead to one process taking over the machine
  - so, we don't do this in modern OS's



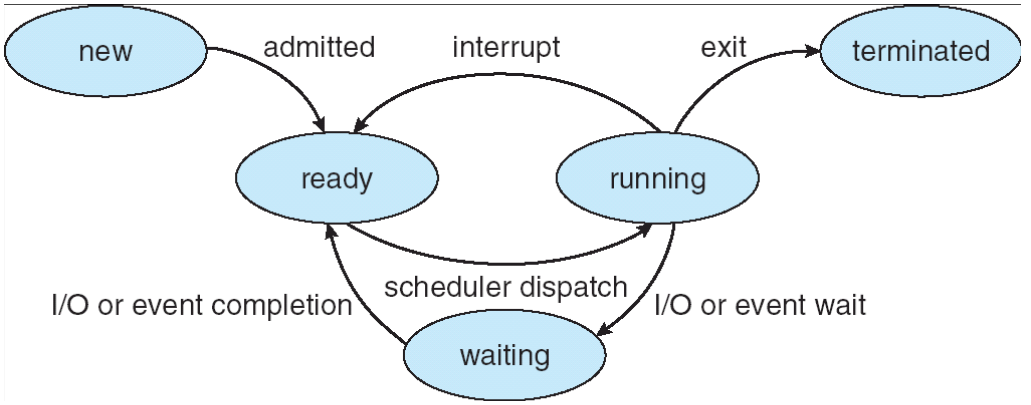
True Multi-Tasking

- Guarantees OS regains control periodically
- Generate HW interrupt periodically
- Timer interrupt cannot be ignored
- Must keep track of processes

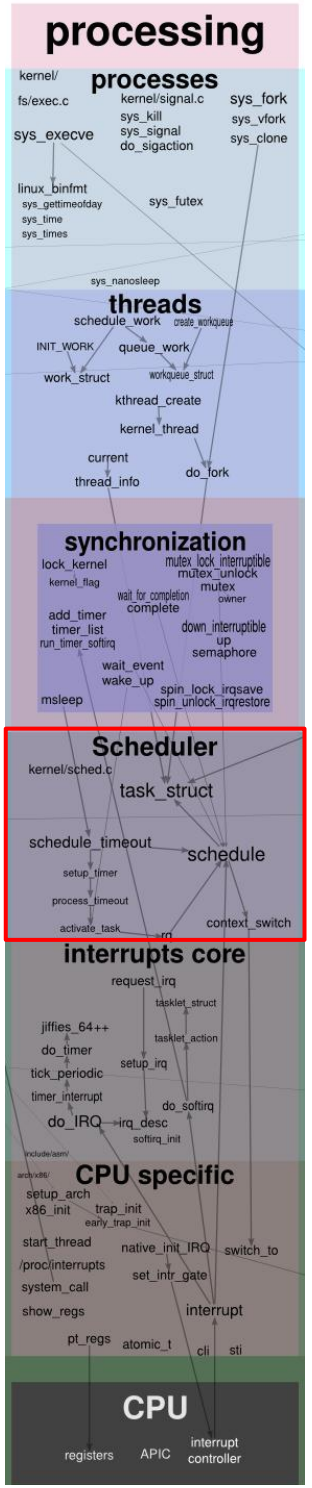




Scheduling

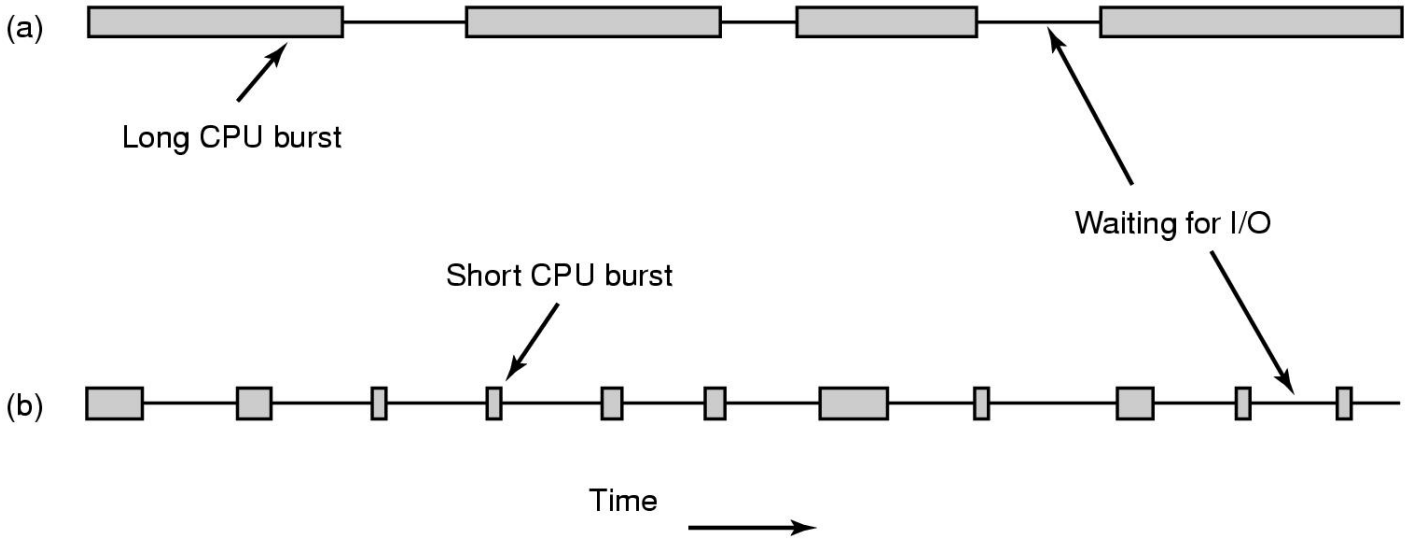


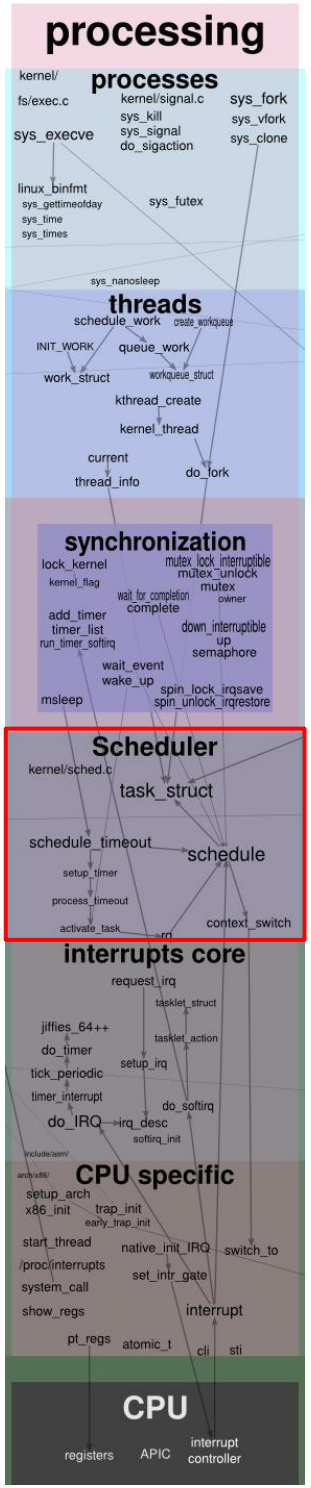
- Goals
  - maximize CPU utilization (reduce execution time)
    - throughput, latency
  - fairness (everyone should run, not wait forever)
- Perform scheduling operations:
  - @ new process creation
  - @ process exit
  - @ process block
  - preemptively (interrupt with timer)
  - non-preemptively (let process run until blocked or finished… probably not a good idea)



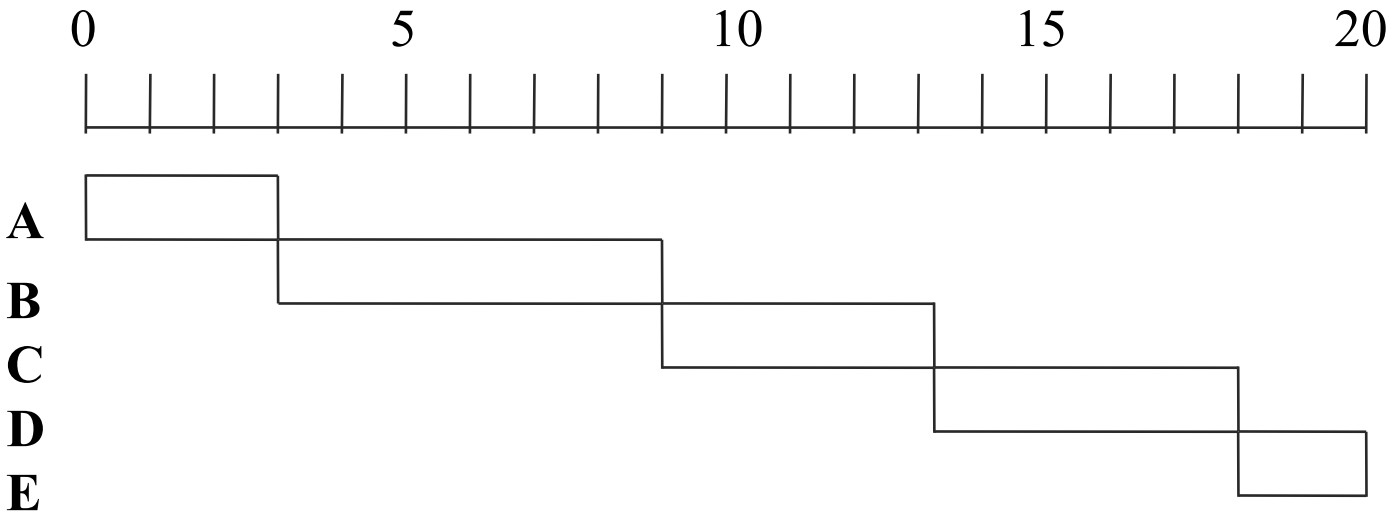
Scheduling (why it works)

- CPU bound vs I/O bound processes
  - opportunity!



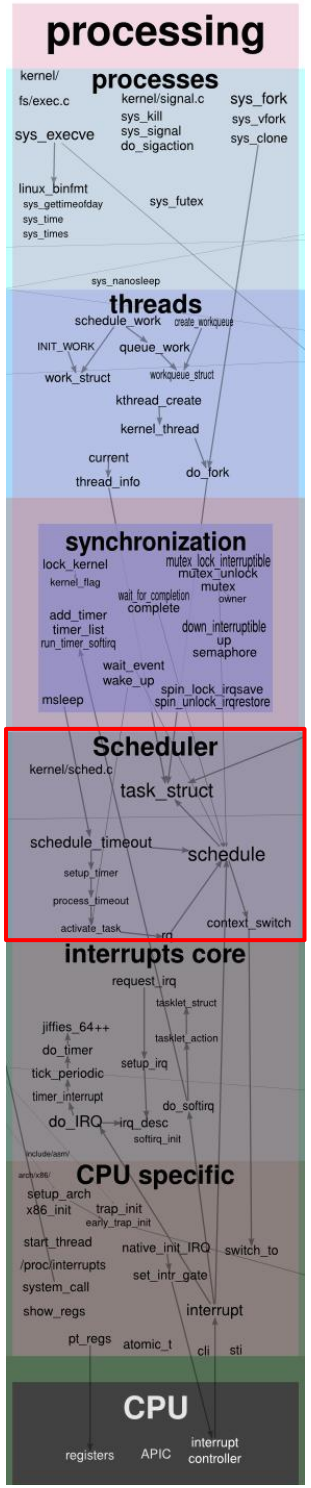


Batch: First-Come First-Served

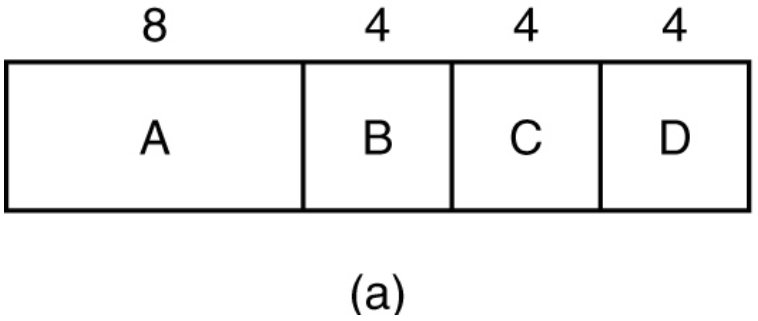


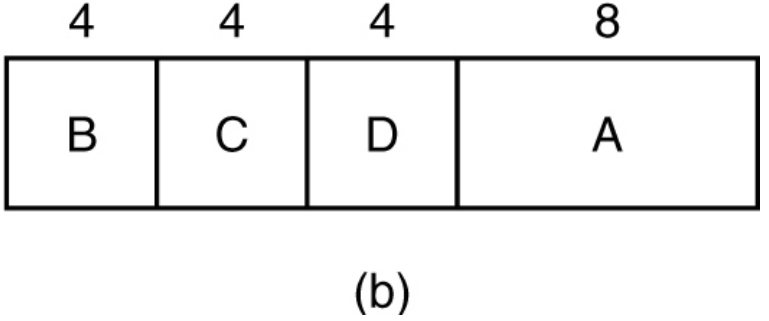
Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

- Nonpreemptive
- Average waiting time is long
- Convoy Effect: A short process may wait a long time to execute

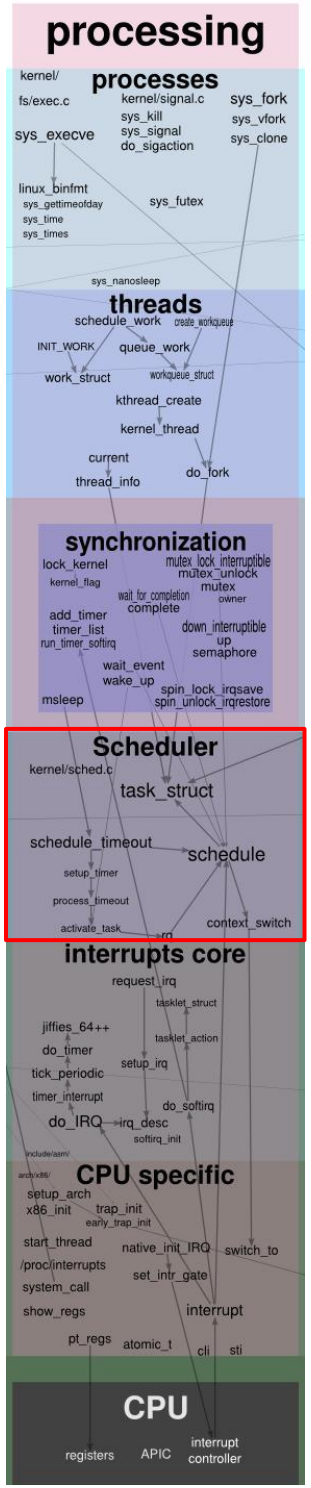


Batch: Shortest Job First



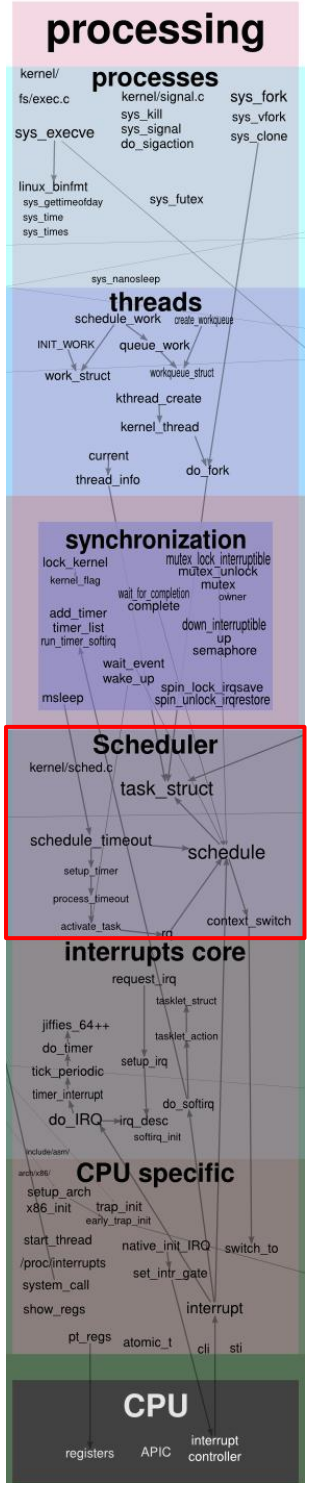


- Schedule process with shortest CPU burst next
- Solves convoy effect
- Optimal when all jobs available simultaneously
  - And when future is known
- Problem?
  - Starvation is possible

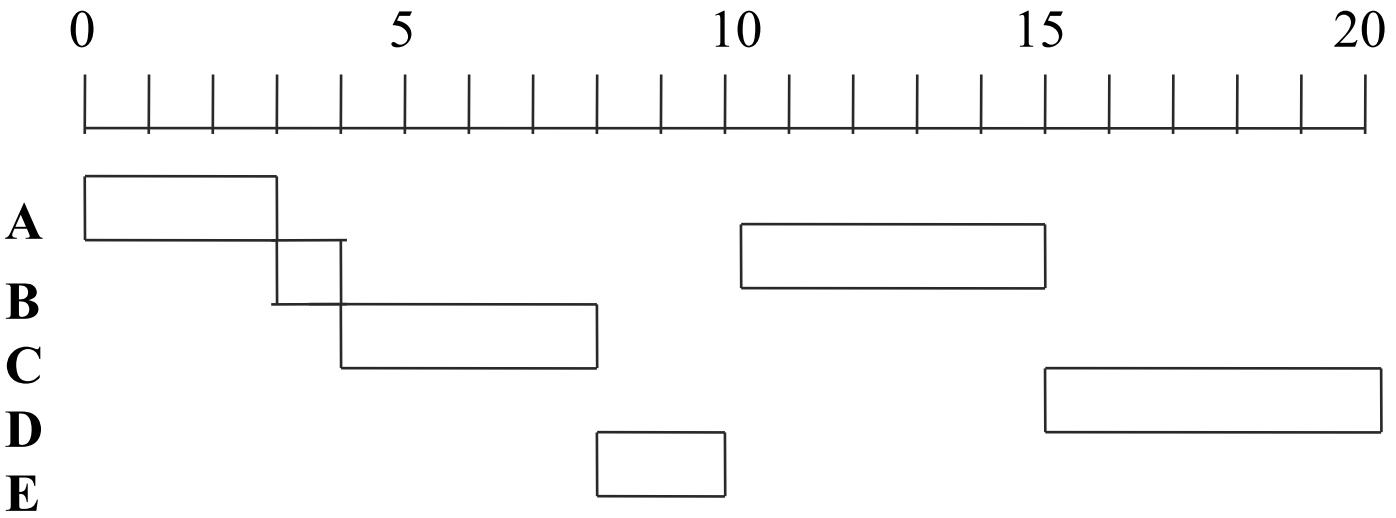


Batch: Shortest Remaining Time

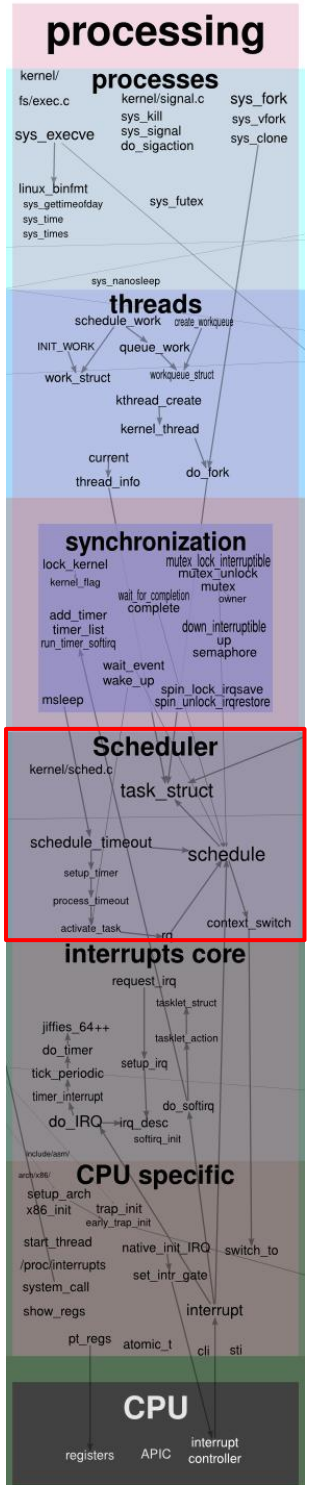
- Choose process whose remaining run time is shortest
- New, short jobs get good service
- Problem?
  - Run-time has to be known in advance



SRTF Example

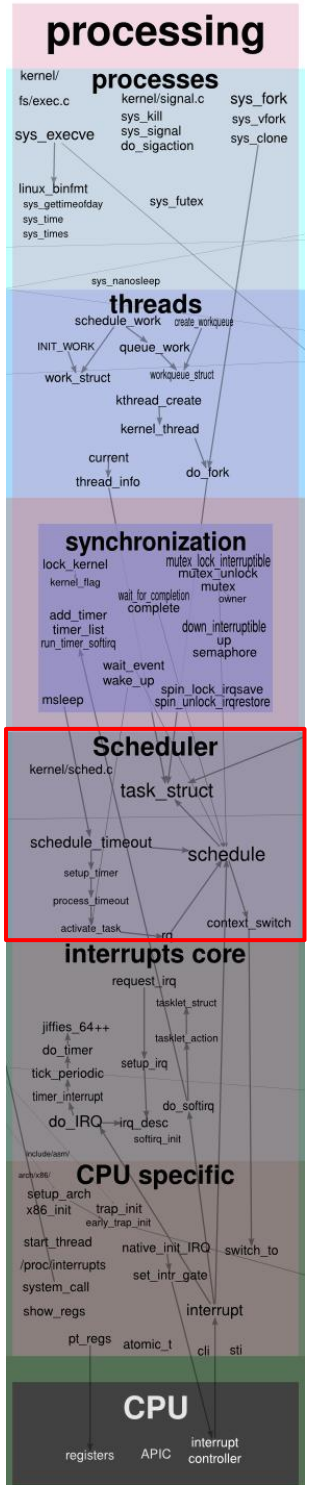


Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2



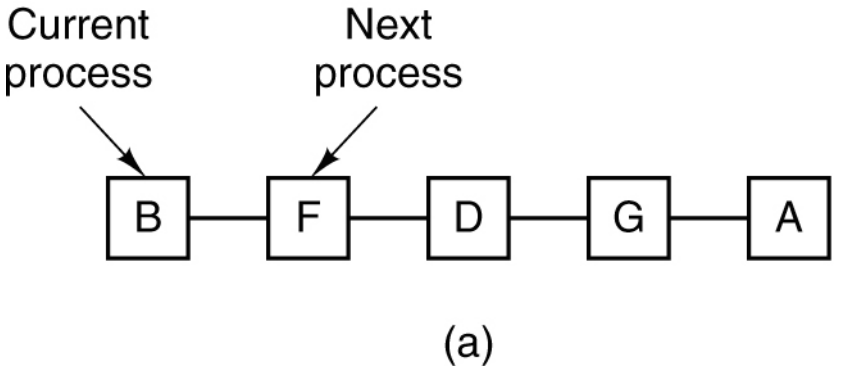
Interactive Systems

- Goals:
  - Response time: respond to users quickly
  - Proportionality: meet users' demands
- Preemption essential
- User actions take precedence

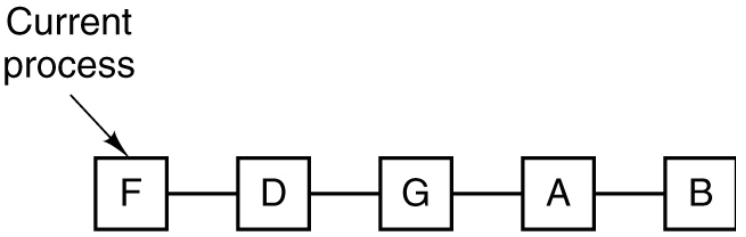


Interactive: Round-Robin

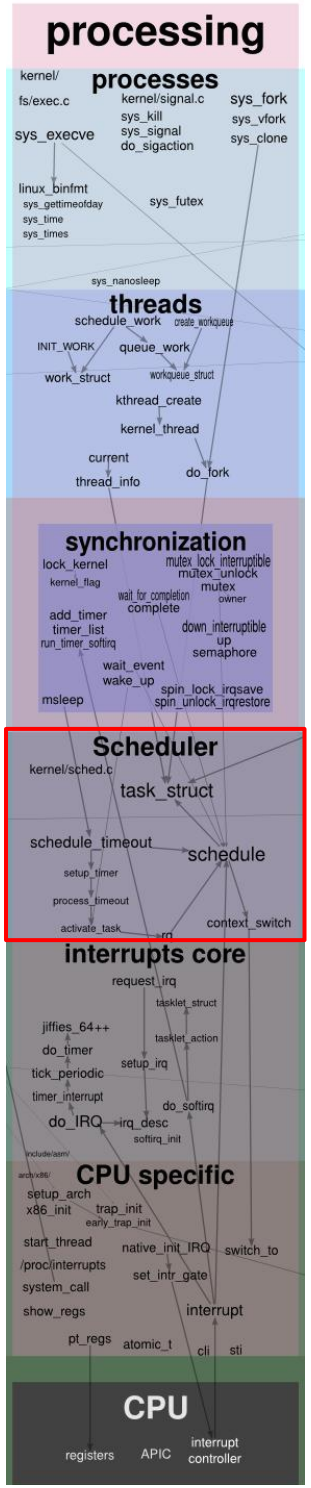
- Each process assigned a quantum
  - Preempted at end of quantum, placed at end of queue
- Short quantum: wastes CPU time
- Long quantum: poor response for interactive tasks



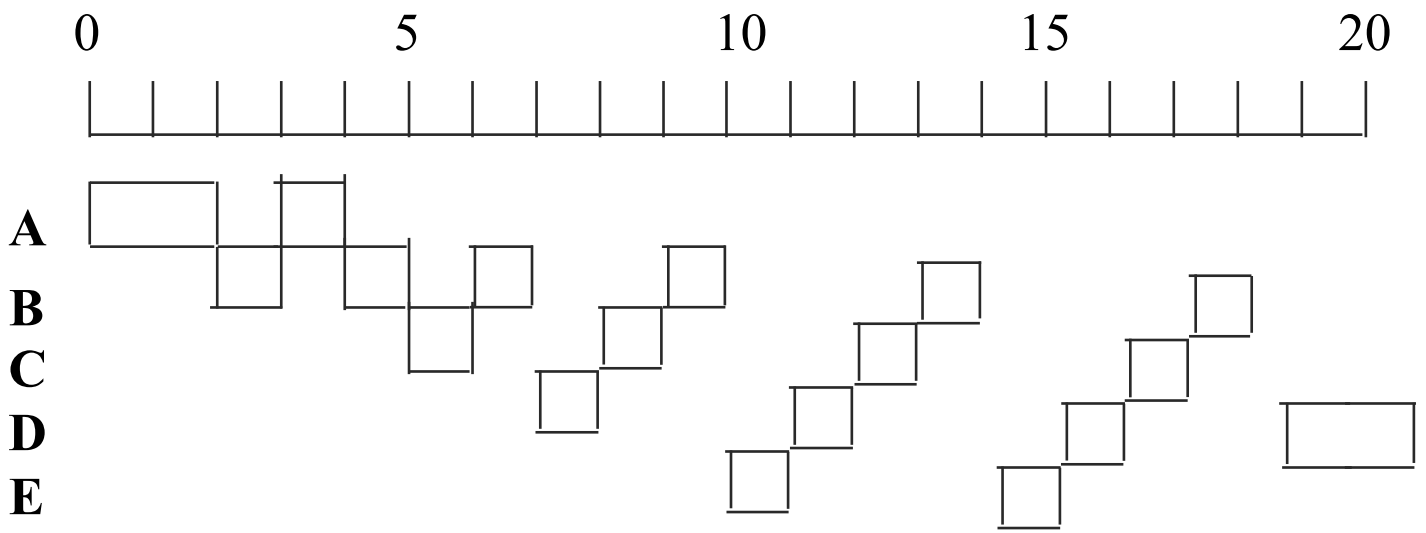
(a)



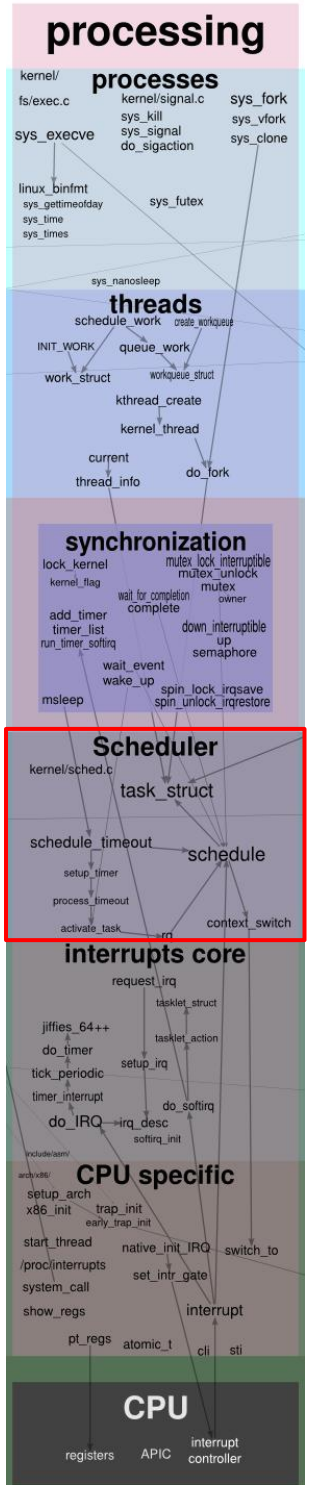
(b)



Round Robin Example

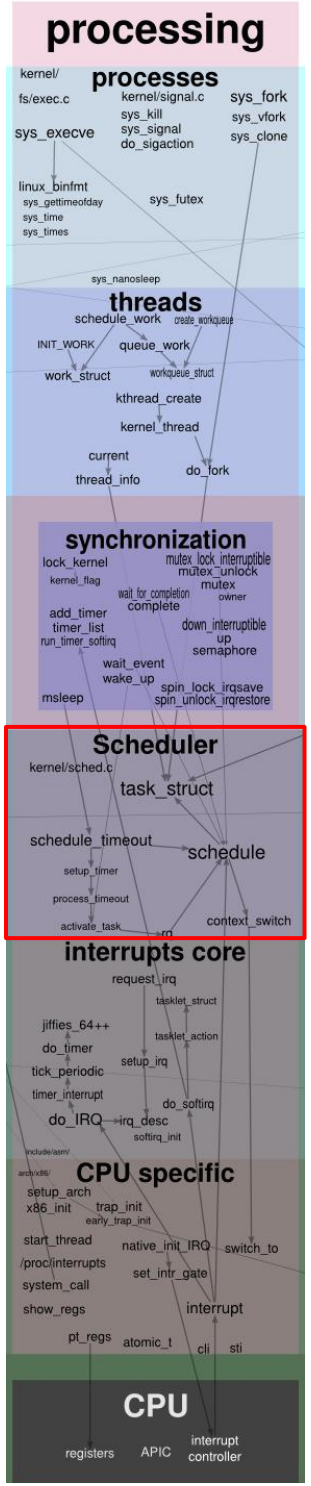


Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2



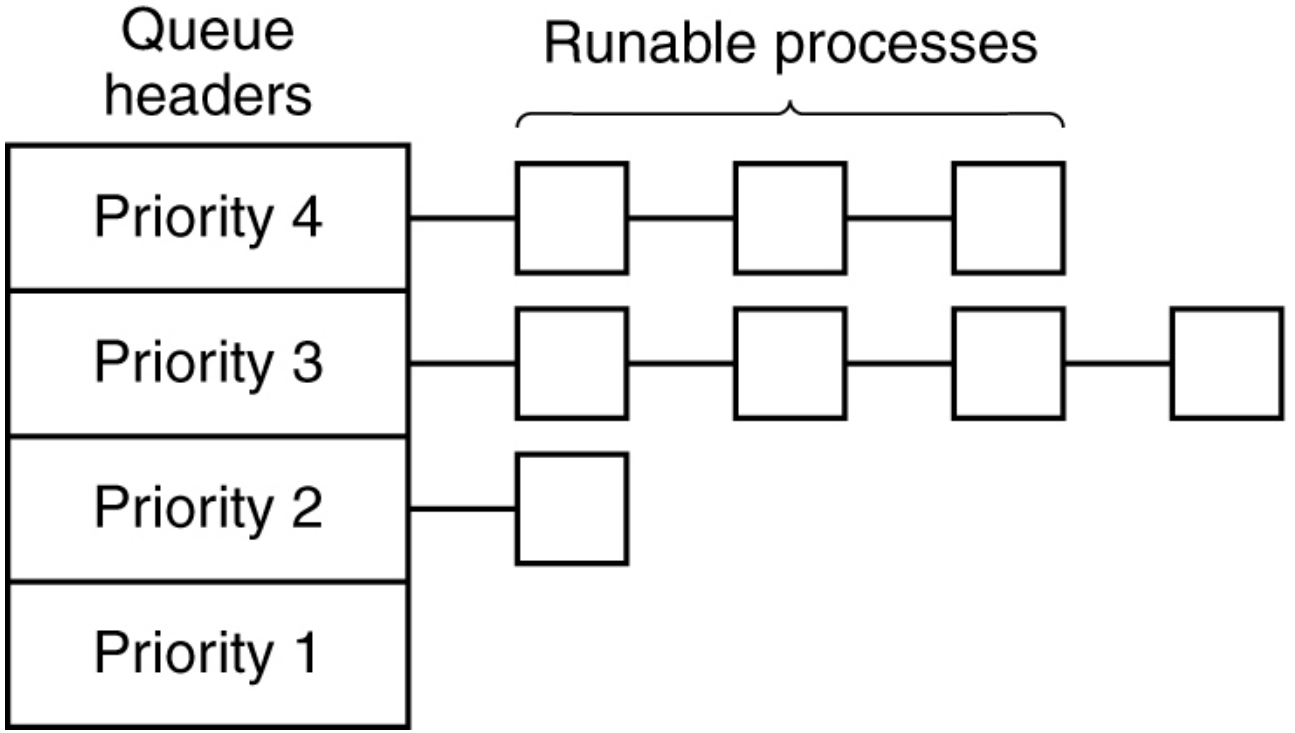
Interactive: Priority Scheduling

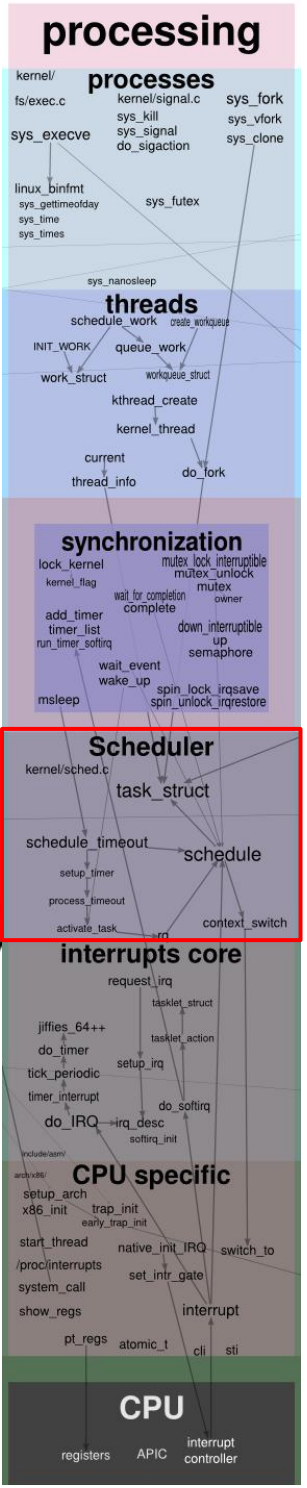
- Each process assigned a priority
  - CPU assigned to highest priority (lowest integer)
- Can work with preemptive or nonpreemptive
- Example: SRJF: shorter runtime= higher priority
- Problem?
  - Starvation: low priority may never execute
- Solution?
  - Aging: processes decrease in priority over time



Priority Scheduling Example

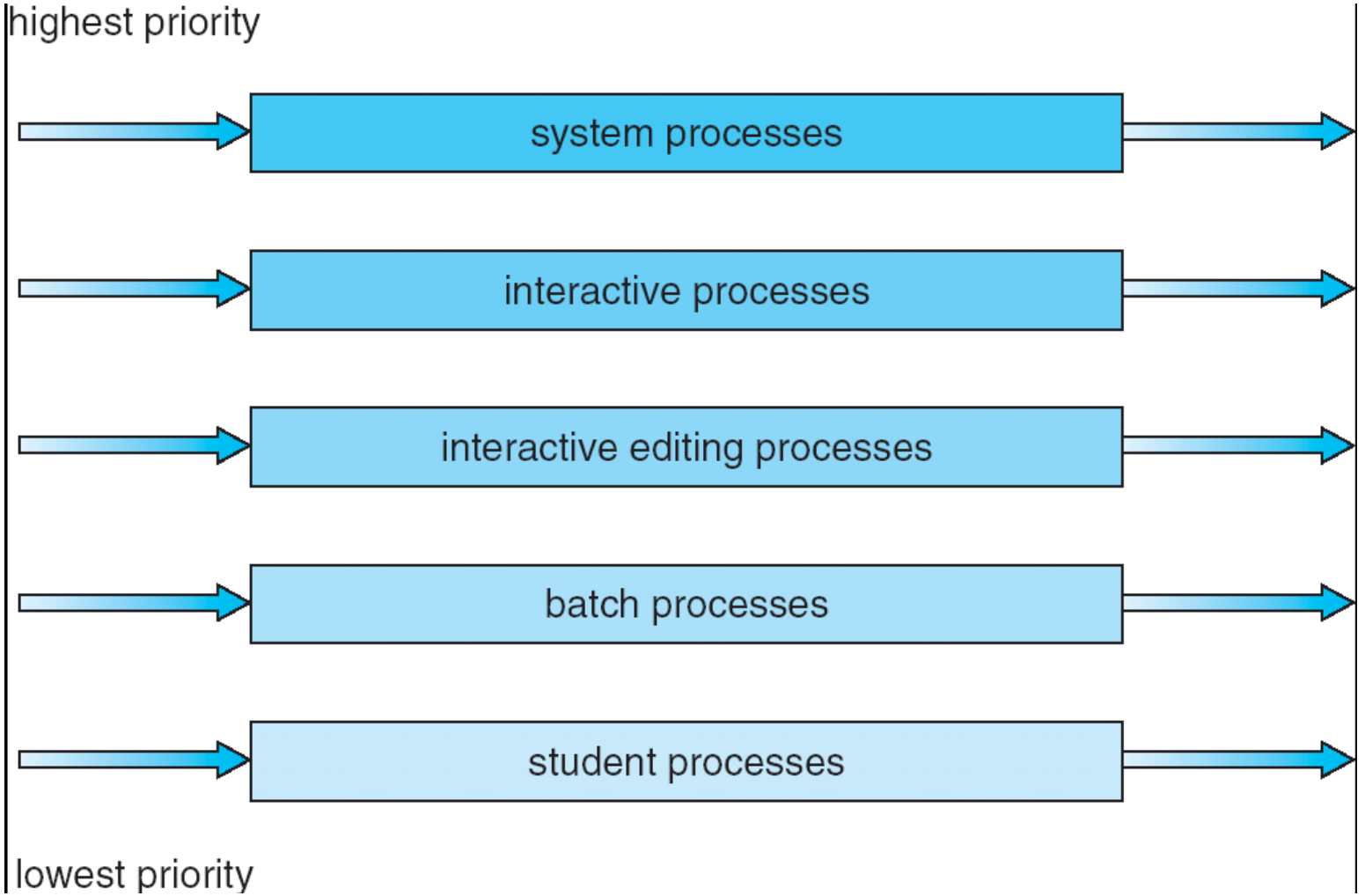
- Round-Robin within classes
- Dynamically adjust priorities to prevent starvation

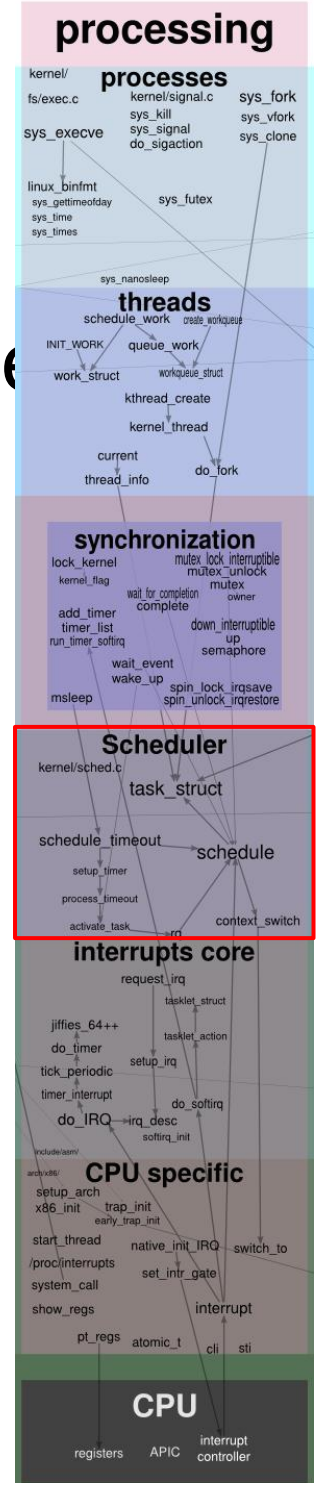




Multilevel queue

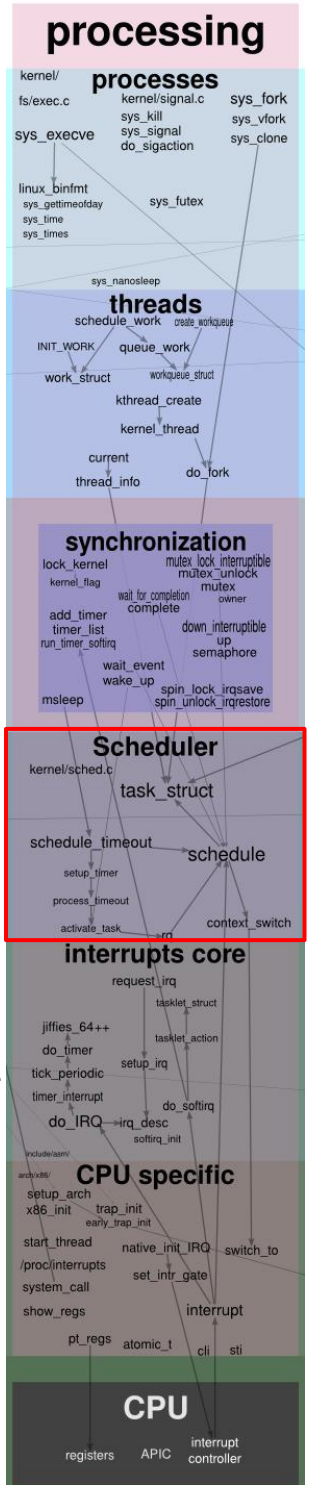
- Ready queue partitioned into separate que





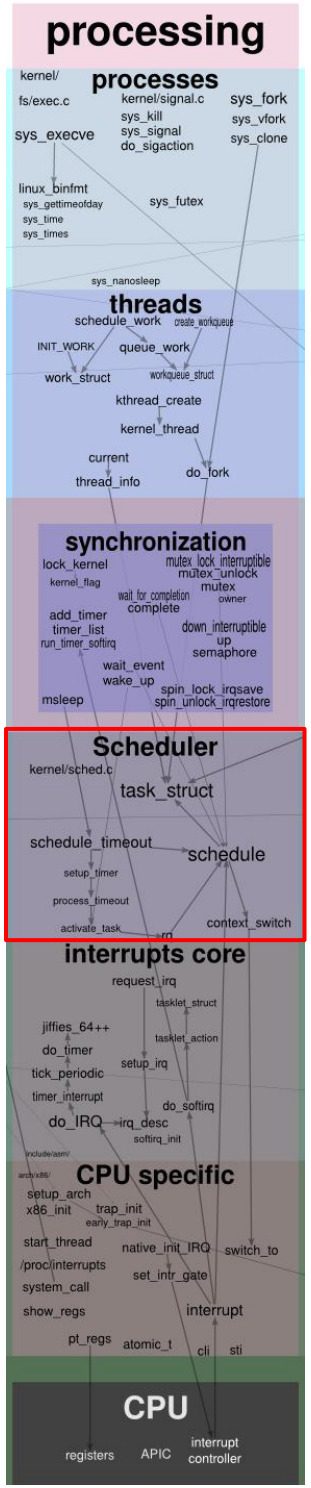
Multilevel queue(2)

- Each queue has its own scheduling algo., e.g.:
  - RR (foreground)
  - FCFS (background)
- Must schedule between queues, too, e.g.:
  - Time slice: each queue gets a certain amount of CPU time for its processes. E.g.:
    - 80% foreground
    - 20% background



Interactive: Lottery Scheduling

- Each process gets a number of “tickets”
- Tickets are called randomly
- Can control the CPU share by allocating specific number of tickets to each process



Interactive: Fair Share

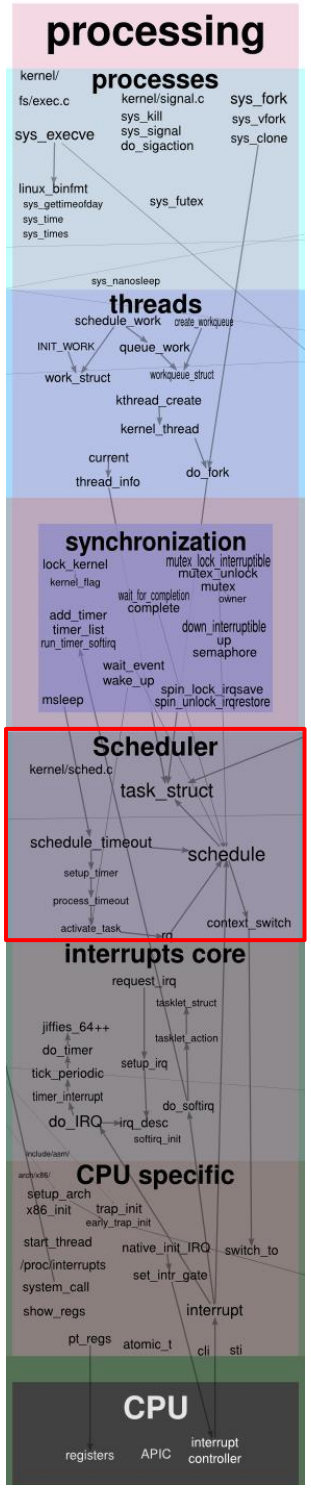
- Round-robin with multiple users may not be fair
  - User with more processes gets more CPU share

  User 1: A, B, C, D

  User 2: E

  Round-robin: A B C D E A B C D E...

  Fair-share: A E B E C E D E...



How are your processes scheduled?

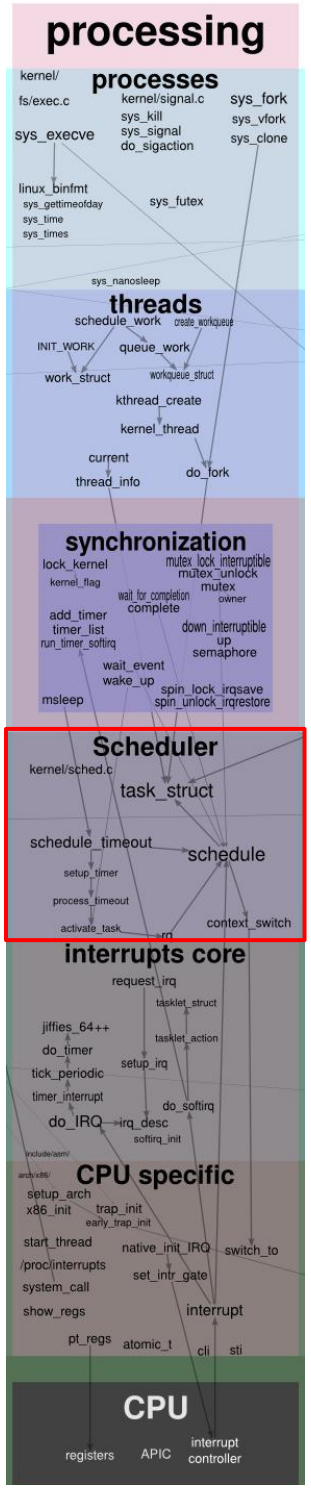
Operating System	Preemption	Algorithm
Amiga OS	Yes	Prioritized Round-robin scheduling
FreeBSD	Yes	Multilevel feedback queue
Linux pre-2.6	Yes	Multilevel feedback queue
Linux 2.6-2.6.23	Yes	O(1) scheduler
Linux post-2.6.23	Yes	Completely Fair Scheduler
Mac OS pre-9	None	Cooperative Scheduler
Mac OS 9	Some	Preemptive for MP tasks, Cooperative Scheduler for processes and threads
Mac OS X	Yes	Multilevel feedback queue
NetBSD	Yes	Multilevel feedback queue
Solaris	Yes	Multilevel feedback queue
Windows 3.1x	None	Cooperative Scheduler
Windows 95, 98, Me	Half	Preemptive for 32-bit processes, Cooperative Scheduler for 16-bit processes
Windows NT (including 2000, XP, Vista, 7, and Server)	Yes	Multilevel feedback queue

Real-Time (RT) Systems

- Goals:
  - Meet deadlines: avoid losing data
  - Predictability: avoid quality degradation

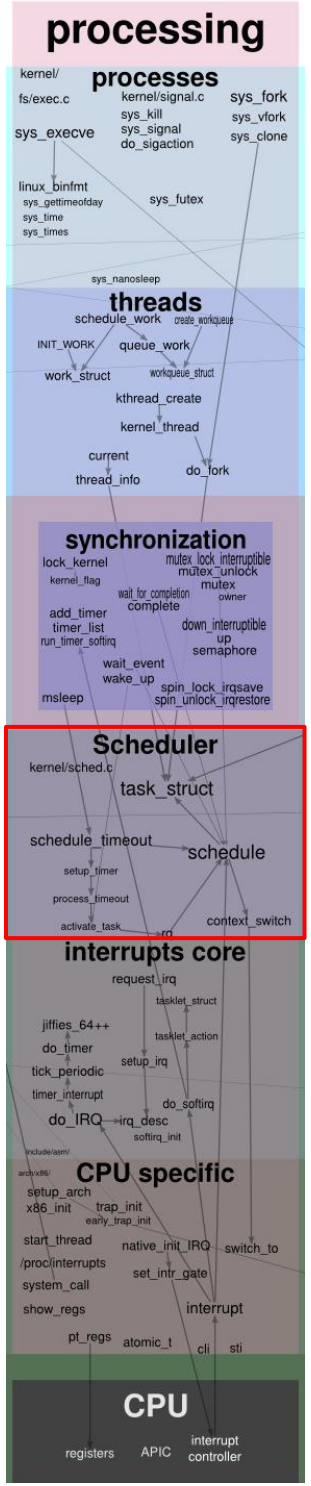
    (e.g. multimedia systems)

- Hard real time: deadlines are absolute
- Soft real time: missed deadlines are tolerable



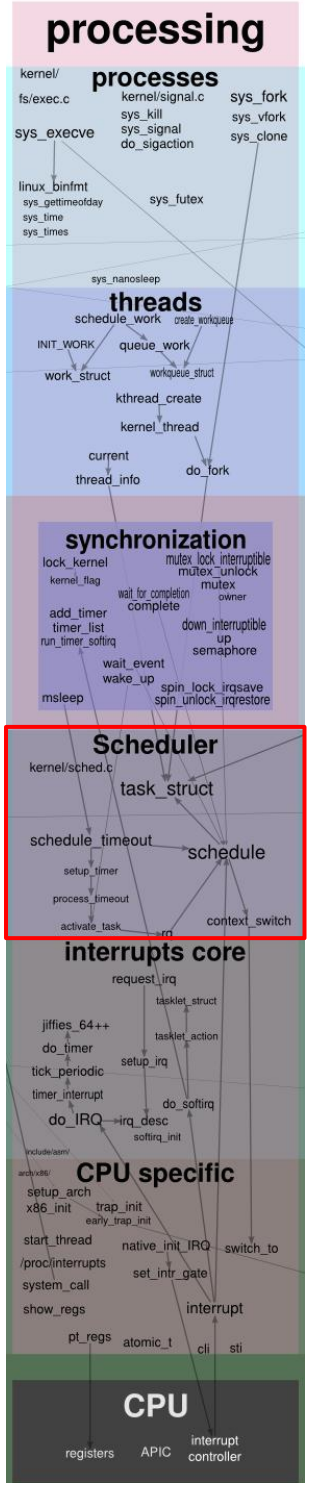
What it means…

- System is deterministic
  - does what/when you expect it to
- Does not mean “fast”
  - but does mean fast worst case times
- Can meet deadlines
  - but you must know what those are



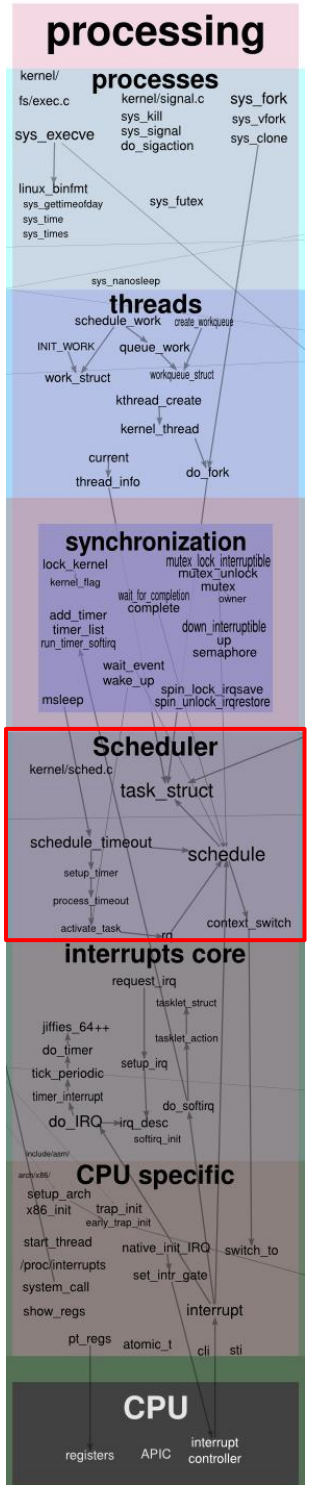
RT usage examples

- Soft deadline example – decoding video:
  - need enough CPU time to finish decoding video at 30 frames/sec
  - release time to deadline: 33.33ms/frame
  - meeting deadline NOT critical
- Hard deadline example – nuclear power plant:
  - meltdown conditions detected?
  - corrective action must be taken immediately must complete within a minute
  - meeting deadline IS critical
    - computation after deadline is useless



Typical RT Systems

- Priority-based scheduler
- Guaranteed maximum time to service interrupts
- Can ensure processes stay memory resident
- Consistent/efficient memory allocation
- Preemptible system calls/preemptible kernel

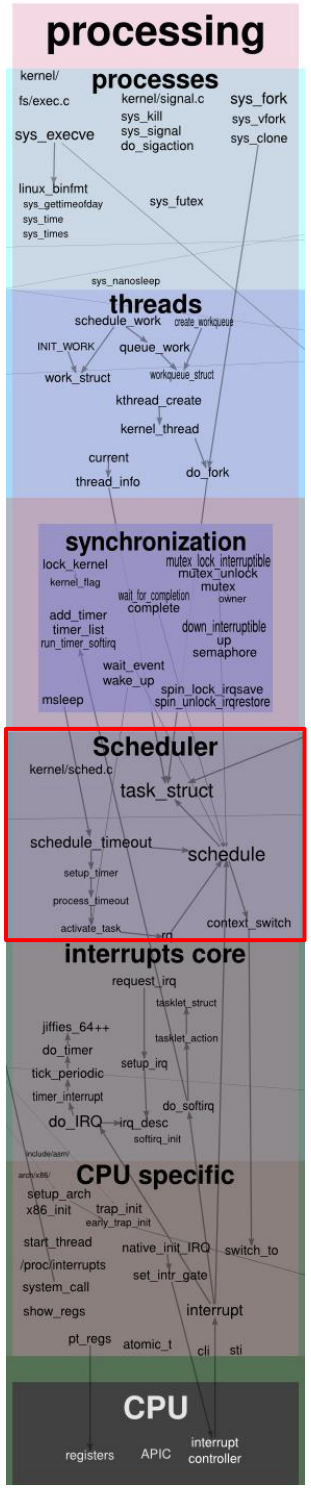


Real-time process types

- Terminating process
  - runs and exits
  - deadline = time to exit
- Non-terminating process
  - periodic – e.g. video decoder producing 30fps
  - aperiodic – occurs in response to some event

(man 7 sched)

```
arrival/wakeup                                  absolute deadline
      |     start time                                |
      |         |                                     |
      v         v                                     v
-----x--------xOOOOOOOOOOOOOOOO--------x--------x---
              |<- comp. time ->|
      |<------- relative deadline ------>|
      |<-------------- period ------------------>|
```

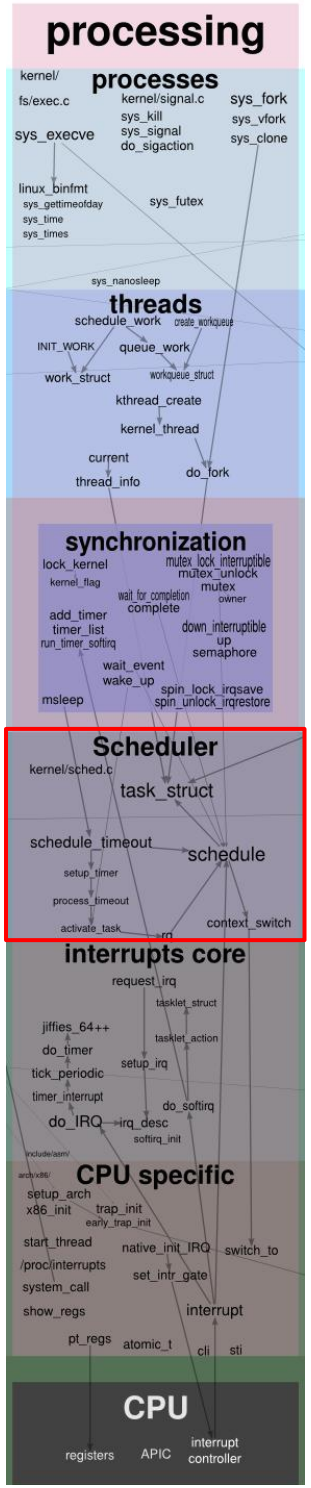


Bounded scheduling example

- Have 3 periodic tasks (a, b, c):
  - with periods (P): 100ms, 200ms, 500ms
  - with comp. times (C): 50ms, 10ms, 100ms

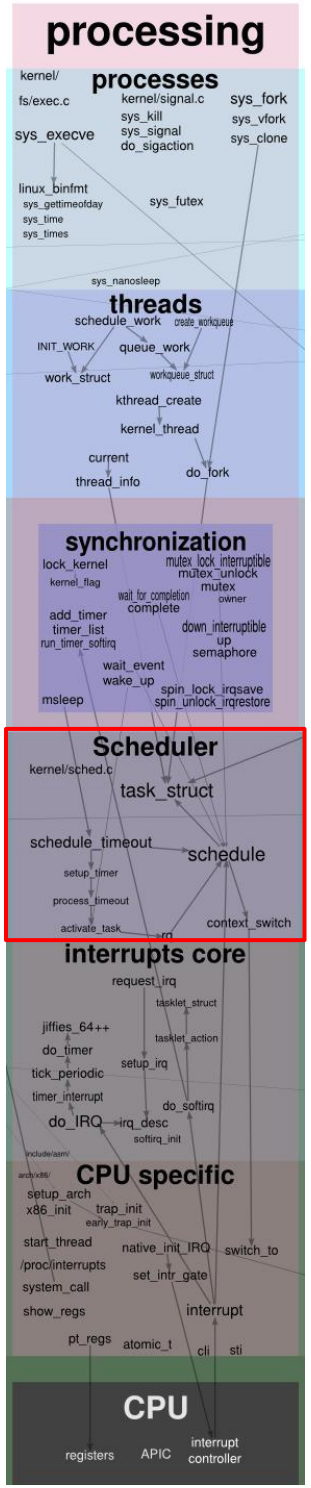
$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leqslant 1$$

- System is schedulable, because
  - .05/.1 + .01/.2 + .1/.5 = .75
- Will it remain schedulable if we add another task with P = 500ms?
  - how would this task's computation time (C) be constrained?



Need a RT OS?

- Linux with RT Patch
  - see: http://events.linuxfoundation.org/images/stories/slides/elc2013_rostedt.pdf
- Windows CE
- Integrity
  - commercial solution
  - used in military jets, Airbus A380
- Many others…



Tour of Linux Schedulers

- Pre-2.4: simple circular queue
- 2.4 : O(n) scheduler
- 2.6 : O(1) scheduler
- 2.6.23 : CFS (completely fair scheduler)

O(n) Scheduler

- Time divided into epochs
  - Periods of time when each task is allowed to exhaust its timeslice
- Timeslices computed at the beginning of each epoch
  - Hence O(n)
  - Nice value = 200ms timeslice
  - Counter value stores number of ticks remaining in alotted timeslice
  - At end of epoch:

    p->counter=(p->counter >> 1) + NICE_TO_TICKS(p->nice)

O(n) Scheduler(2)

- So, a process gets more time in next epoch if it hasn't exhausted its timeslice last time
  - Back to base timeslice if suddenly CPU-bound
- Parent's timeslice shared with children
- Next task scheduled by goodness()

```
if(p->policy != SCHED_NORMAL)  //indicates RT task
    return 1000+p->rt_priority;
if(p->counter==0) return;
return p->counter+p->priority;
```

O(n) Problems

- Scalability
  - Overhead increases with tasks
- Large average timeslices
  - Default is 200ms, average is 210ms
- Sleepy tasks get huge bonuses
- Above and nonpreemptable kernel spell trouble for RT tasks

O(1) Scheduler

- Goals
  - Good interactive performance under high load
  - Fairness and priorities
  - SMP efficiency and affinity
- How
  - Gets rid of loops, no recalculation, no goodness
  - per-cpu runqueues and locks

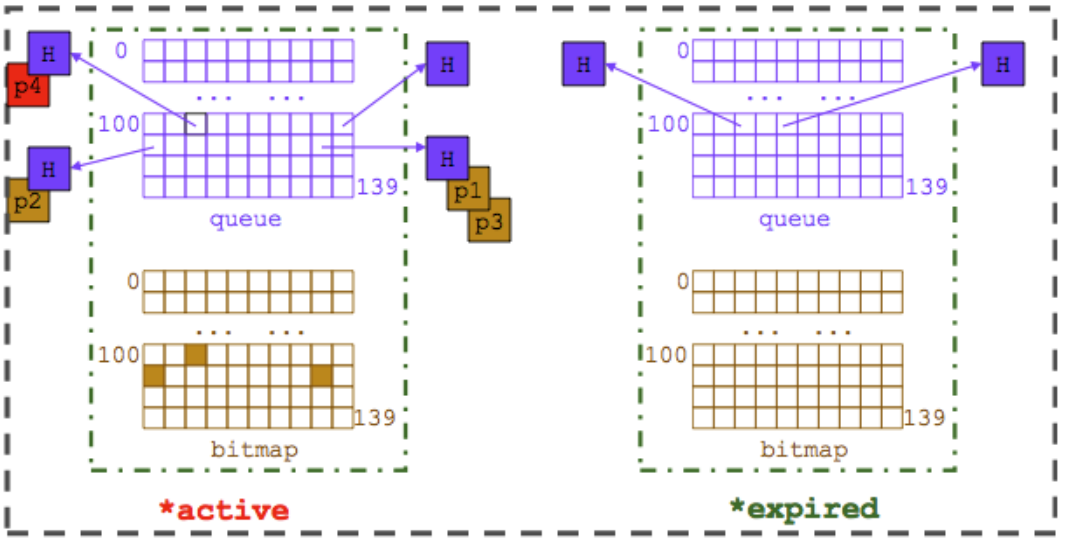
O(1) Scheduler

- Two priority arrays w/ 140 priority levels
  - Active and expired array (swap when active empty)
  - 0-99: real-time tasks (200ms quantum)
  - 100-140: nice tasks (10ms quantum)
  - Round-robin w/in priority level
- Bitmap indicates whether tasks are waiting in a priority level
- Timeslices calculated when they run out

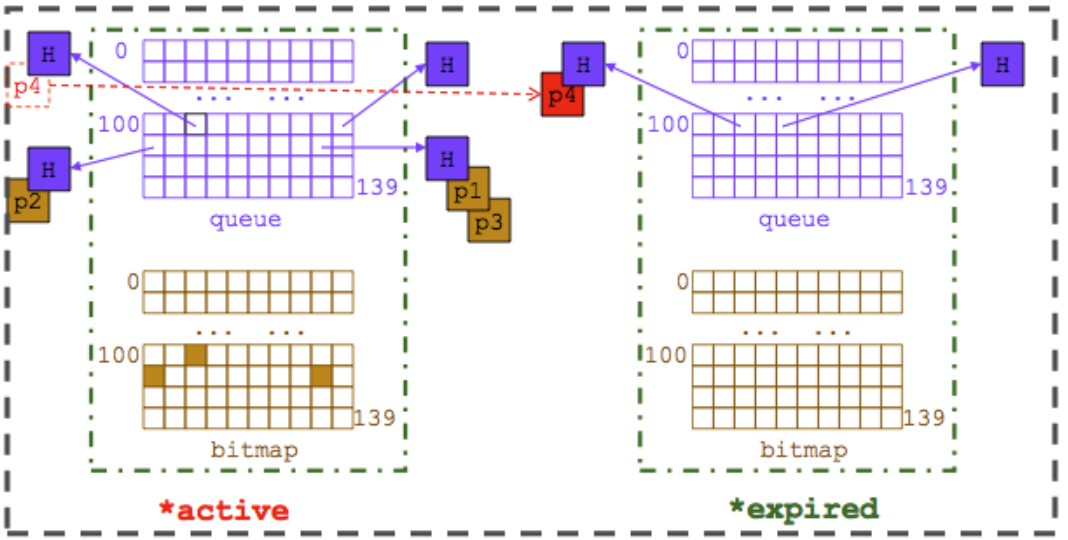
O(1) Scheduler

- All operations over priority arrays are O(1), hence scheduler is O(1)
- Static priority can be set by user
- Dynamic priority reset due to “interactivity” of the task
  - Highly interactive processes don't expire
    - Unless another is starved on expired array
- Default timeslice: 100ms
- Can also have RT and FIFO tasks

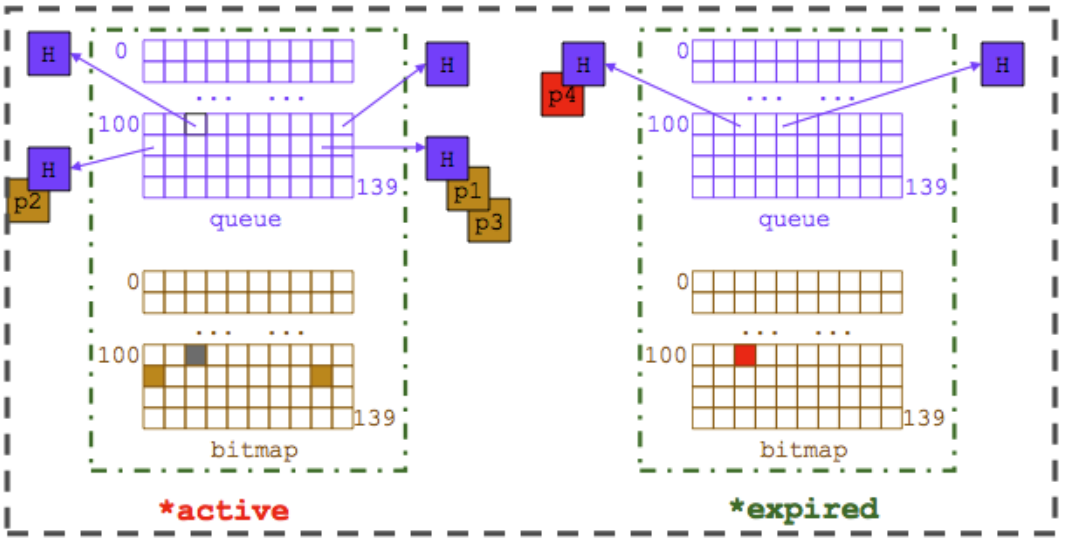
O(1) Example


H
p4
H
p2
0
100
139
queue
H
H
p1
p3
0
100
139
bitmap
*active
H
0
100
139
queue
H
0
100
139
bitmap
*expired

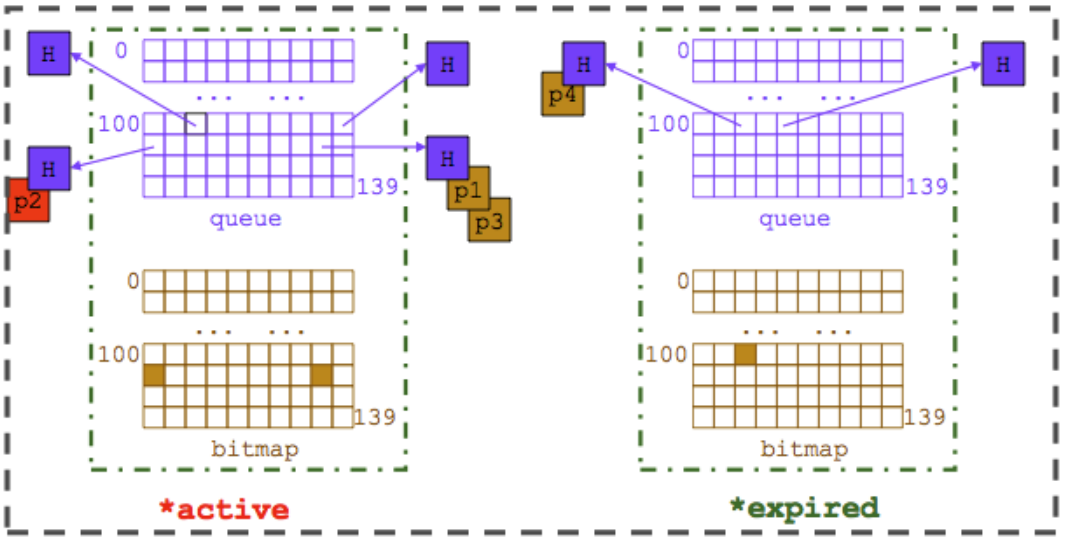
O(1) Example



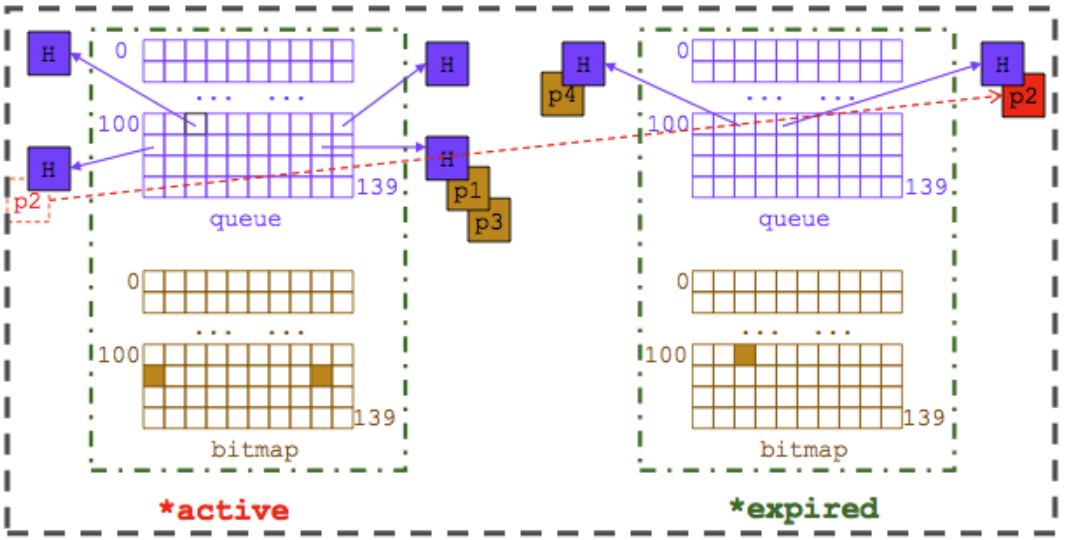
O(1) Example



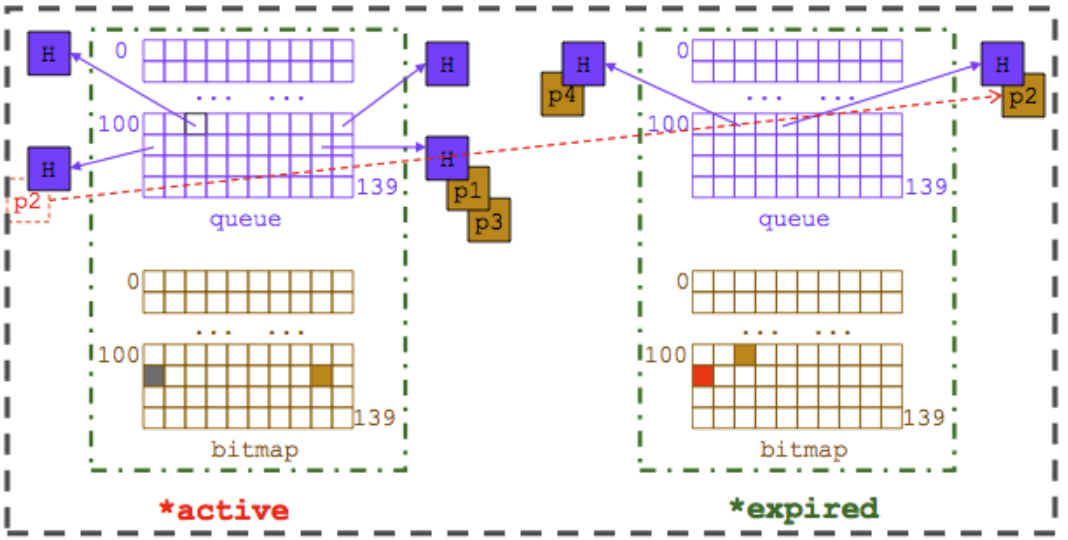
O(1) Example



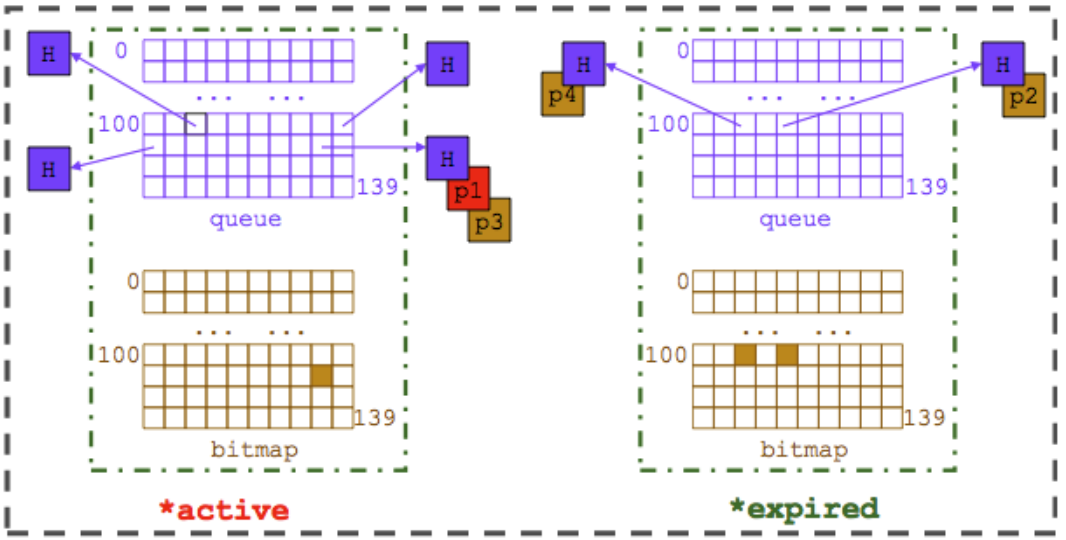
O(1) Example



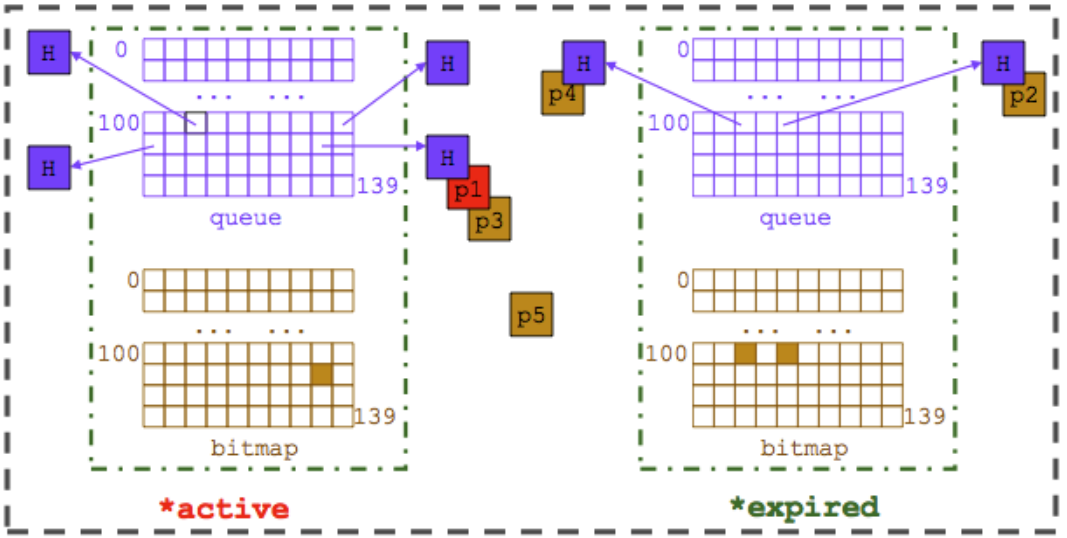
O(1) Example



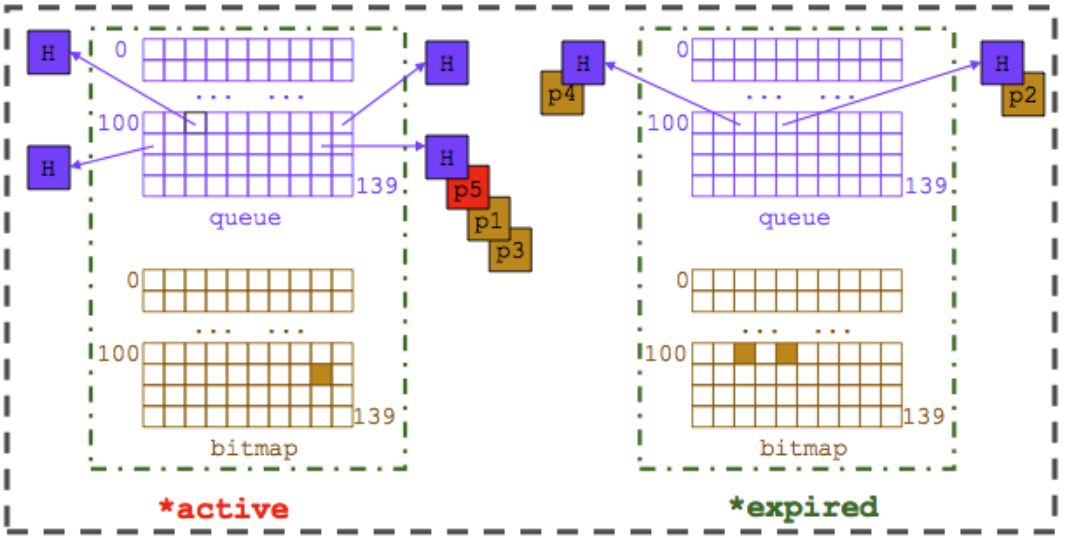
O(1) Example



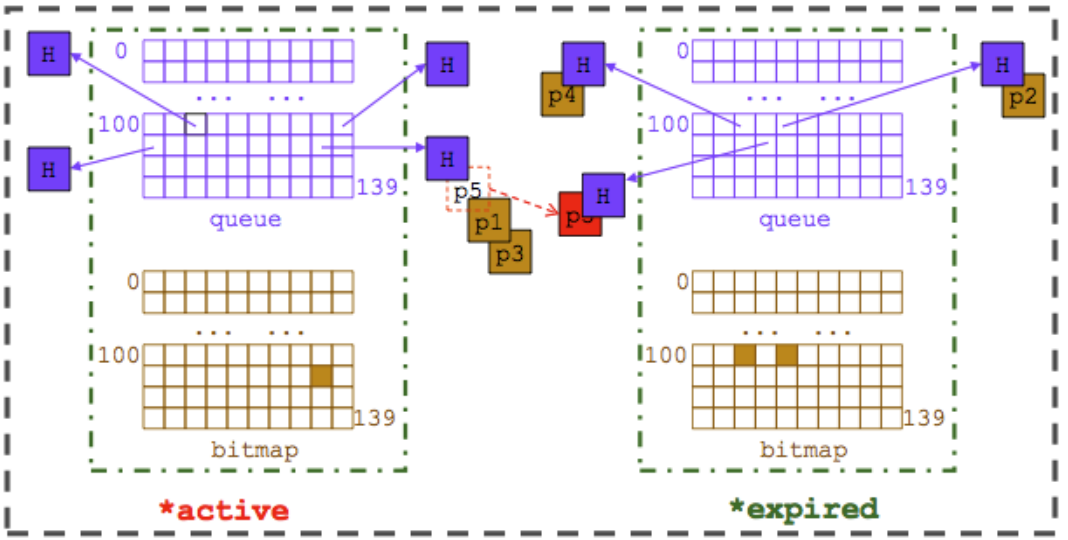
O(1) Example



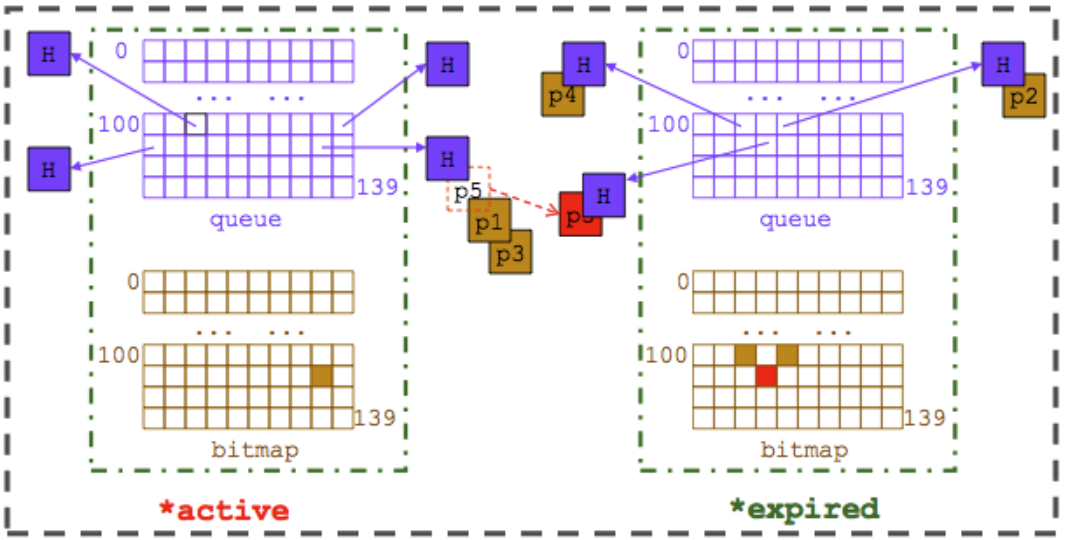
O(1) Example



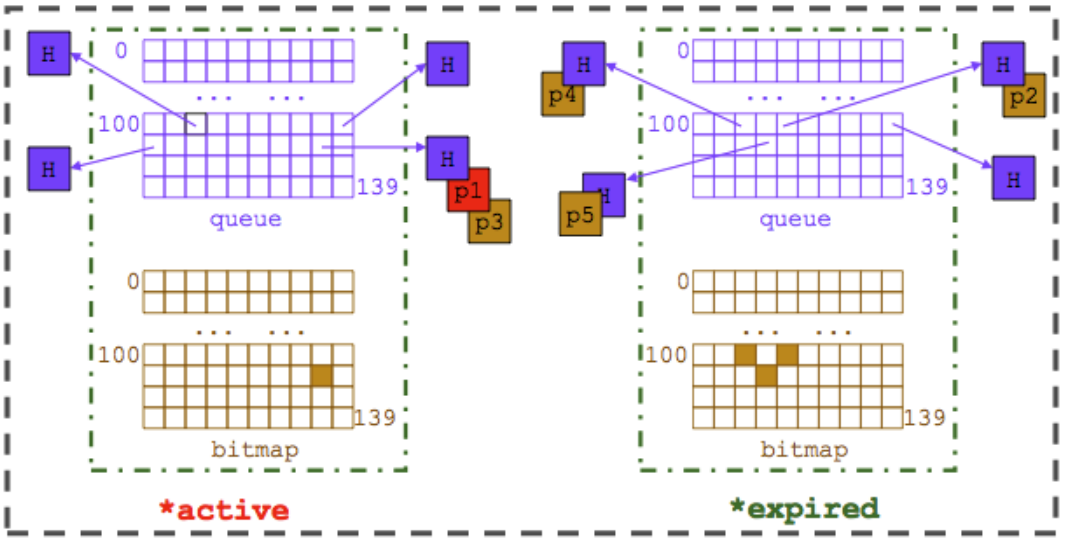
O(1) Example



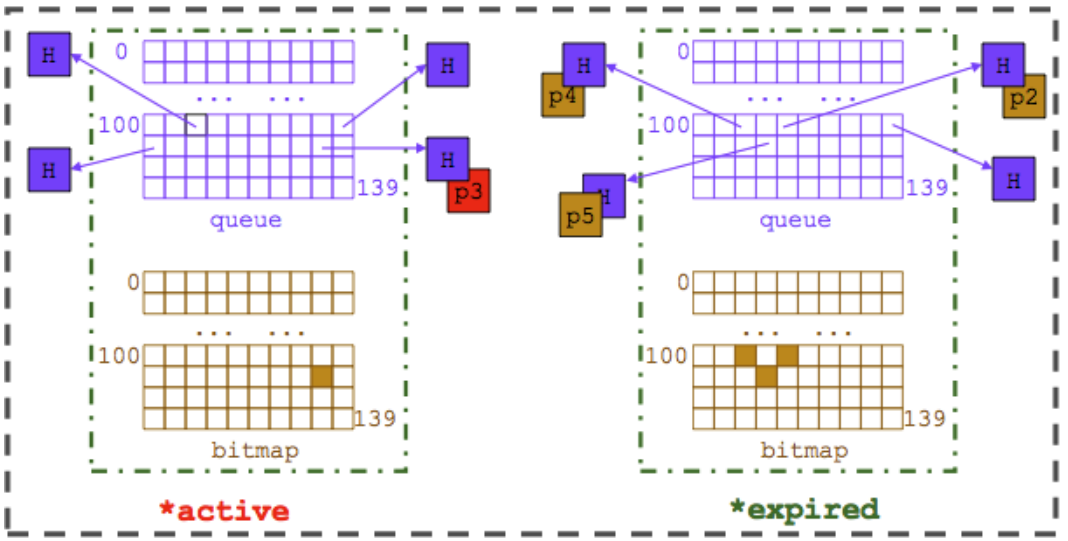
O(1) Example



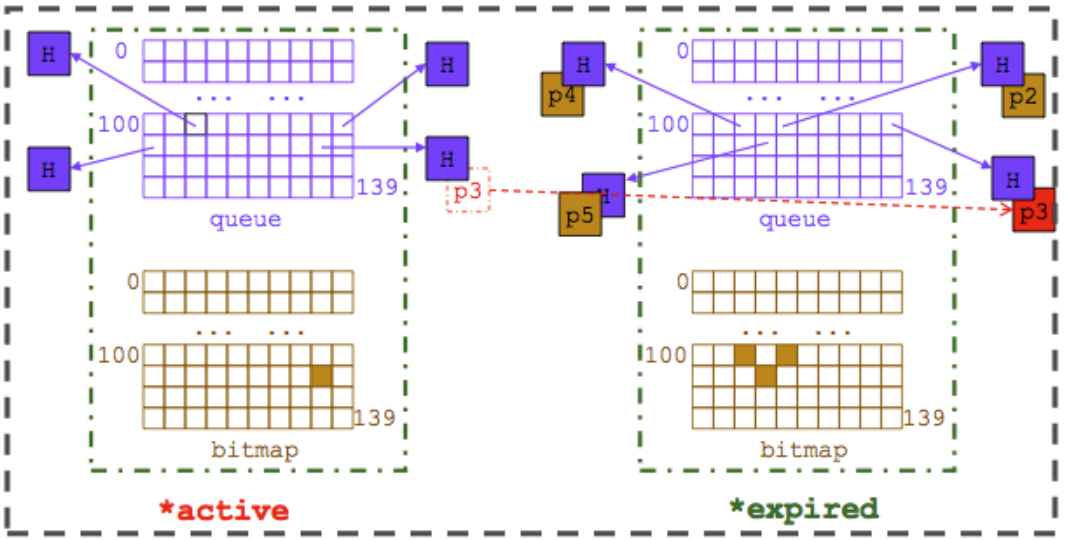
O(1) Example



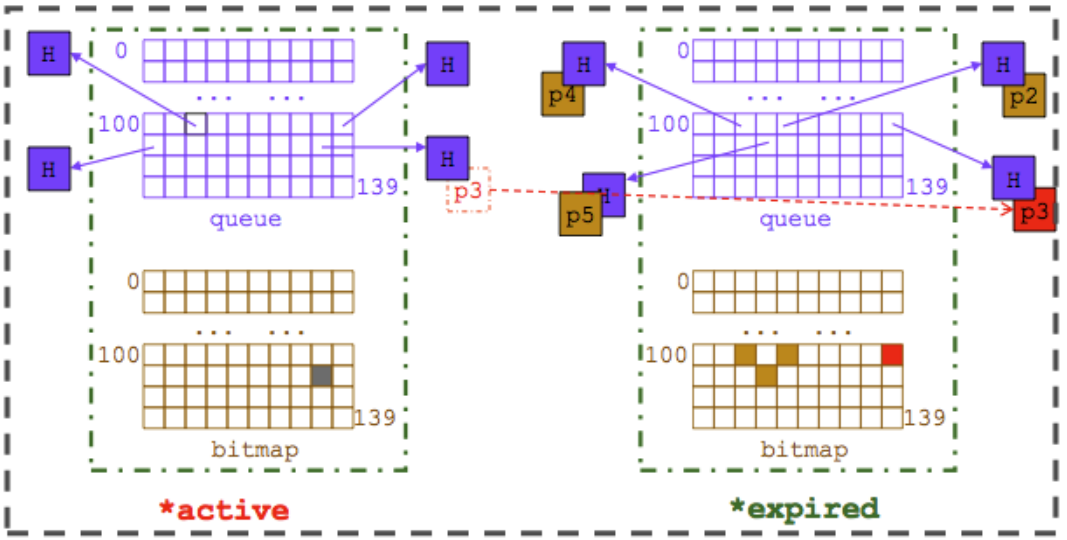
O(1) Example



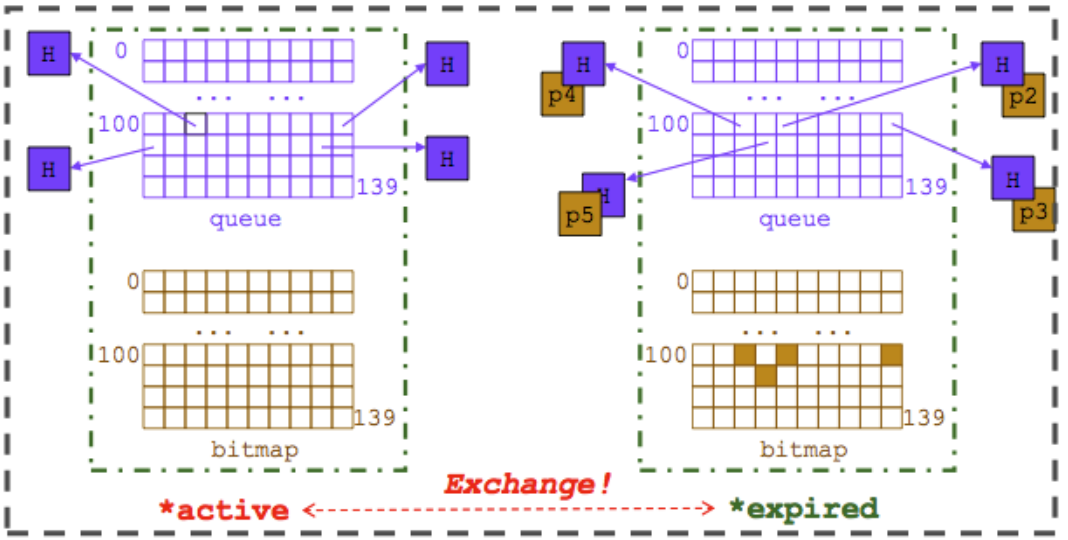
O(1) Example



O(1) Example

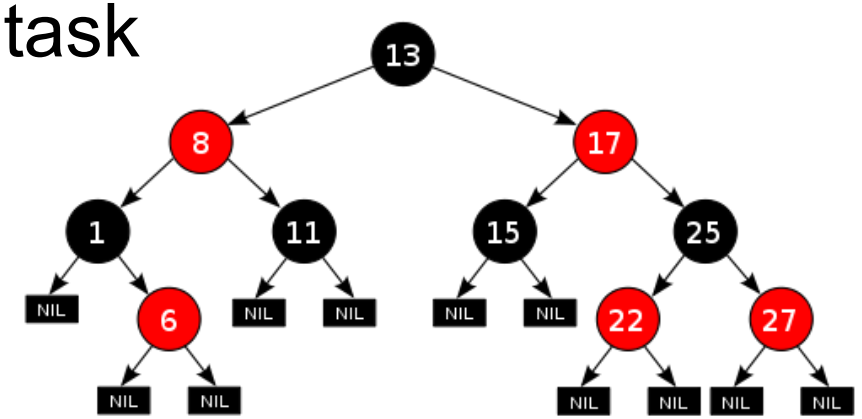


O(1) Example



CFS (Completely Fair Scheduler)

- Implements runqueue in rbtree
  - Rbtree gives worst-case guarantees for insertion/deletion/search
  - Rbtree is time-ordered
  - Choosing a task to run O(1)
  - Reinserting a task into rbtree O(n log n)
- Run left-most task

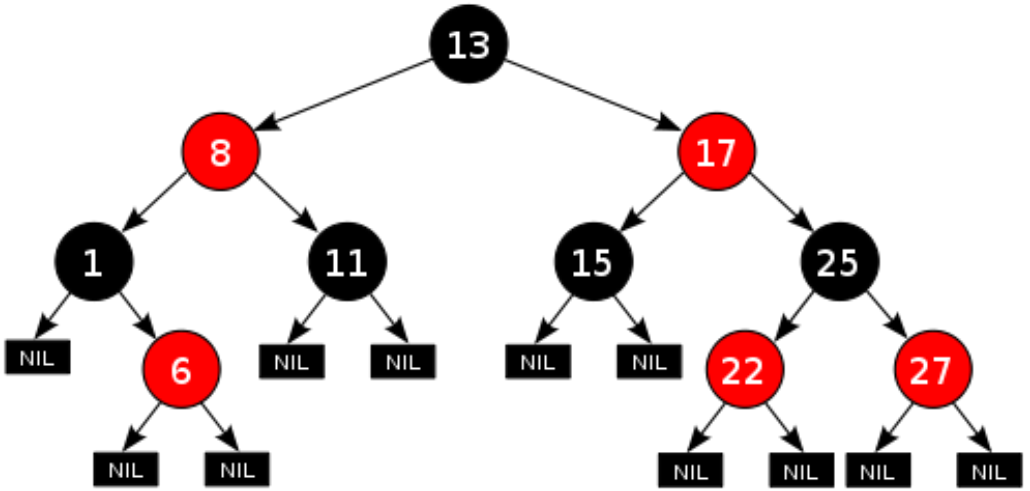


Red-Black Trees

- https://www.cs.usfca.edu/~galles/visualization/RedBlack.html
- Color-coded binary tree
- Each node is either red or black
- All leaves are the same color as the root
- Every red node must have two black children
- Every path from a given node to any of its descendant leaves must contain the same number of black nodes

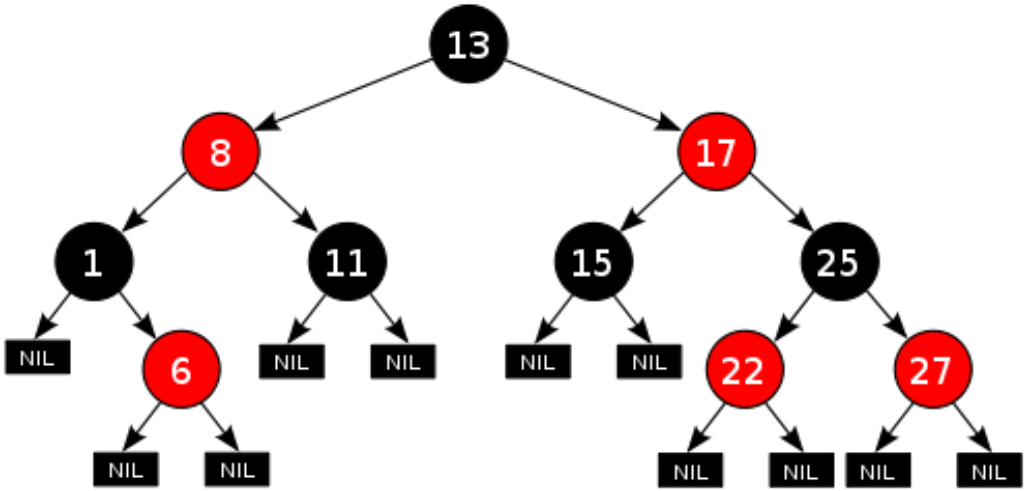
CFS RB-tree Task Placement

- Order tasks by time spent on CPU
  - *vruntime* – ns granularity
  - each node is a task (single runqueue)
  - tasks on left have had less CPU time
  - tasks on right have consumed more CPU time thus far



CFS RB-tree *vruntime* Update

- For running task, increment *vruntime* periodically
  - compare to current left-most task
  - if still smaller, continue running
  - if greater, insert into appropriate place in tree



CFS RB-tree *vruntime* Update/Priorities

- Time passes more quickly for lower-priority tasks
  - i.e. *vruntime* made to increase more rapidly
  - these tasks lose the CPU faster
  - (no separate priority runqueues)
- Per-CPU runqueues
  - affinity – tasks generally stay on same CPU (why?)