20-EECE-4029/6029, 20-ELTN-4022 - **Introduction to Operating Systems**

Review Notes for Exam 1

September, 2013

Operating Systems Concepts

Motivation

- Efficient use of resources to economically manage many processes at a time
- Examples of resources:
  - ▷ Memory - several levels costly-fast to inexpensive-slow
  - ▷ Devices - network card, USB devices, hard drives
  - ▷ Structures - work queues, task lists
  - ▷ Hardware caches
  - ▷ Software library functions
  - ▷ Threads/Processes/Processors
- Provide services for applications
  - ▷ File system - open, close, read, write etc.
  - ▷ Network communication
  - ▷ Interface to hardware's graphics/sound functions
  - ▷ System calls

Processes

- Set of instructions, memory from heap, stack, stack pointer, program counter, registers
- May compete for resources - deadlock, livelock, starvation can result
- May spawn threads
- May be interrupted
- May communicate and cooperate with other processes
- May transfer data to/from devices

Kernel space - User space

- Kernel space (linux) has the following restrictions:
  - ▷ The kernel is limited to about 1GB of virtual and physical memory
  - ▷ The kernel's memory is not pageable
  - ▷ The kernel usually wants physically contiguous memory
  - ▷ Often, the kernel must allocate the memory without sleeping
  - ▷ Mistakes in the kernel can be extremely costly

Context

- **Kernel context** - kernel runs system calls on behalf of some application. Interrupts may occur in kernel context. Context is process control block, device information, queues.
- **Interrupt context** - entered when an interrupt is generated (e.g. by `procfs`). Interrupts are turned off. Context is the memory of what the kernel was doing when it was interrupted.
- **User context** - state of normal application when interrupted, say by the end of a time-slice.

Operating Systems Concepts

Procfs

- Provides communication between user and kernel
- `open`, `release`, `read`, `write` are basic operations that are entered via interrupt.
- Due to kernel restrictions, `copy_to_user` and `copy_from_user` are used to transfer data between kernel space and user space.

Memory Hierarchy

- Fast memory is expensive, cheap memory is slow.
- There are many types of memory that fall between the above extremes:
  - ▷ Registers - there are relatively few of these, each might be at most 8 bytes long.
  - ▷ Hardware cache - cache lines of about 64 bytes each, total of about 64KB, save lookups to slower main memory
  - ▷ Translation Lookaside Buffers - page table caches of about 4MB each for quick translations from virtual to physical addresses
  - ▷ There might be two levels of the caches mentioned above
  - ▷ RAM operates more than 10 times slower than the caches in the processor but GBs of RAM may be available at reasonable cost.
  - ▷ DISK access time may be as long as .1msec
  - ▷ The cloud has gobs of space but uncertain reliability and access time, not to mention potential security problems.

Memory Management

- Efficiently allocate and free segments of memory to/from processes that request it taking into account all processes that are running or runable.
- Provide virtual memory addresses to processes, making effective use of the entire memory hierarchy - the virtual addresses may overlap between processes.

IO - Devices

- Device drivers
- IO Channels
- Channel controllers
- DMA transfers

File system

- Subsystem that controls how data is stored and retrieved
- Organization depends on the application but access methods are roughly the same
- Hierarchical file system
- Reliability improvements through redundancy

Processes and Threads

Process

- An instance of a running program that maintains a changing state consisting of instructions, register values, stack, heap space, a program counter, stack pointer, a frame pointer, and flags.
- May be interrupted (suspended) and resumed
- May need resources such as memory, hard disk to complete
- May be active along with many other processes
- Is in one of the following at any time: running, ready to run, blocked but capable of running.

Thread

- A procedure that runs independently from its main program
- Has its own stack

Inter-process control

- Locks

  ▷ **Semaphore** - May be used in critical sections that go to sleep - unlike the spinlock. The semaphore is just a counter which may be incremented or decremented. Unlike the `mutex` this is not a locking function (that is, threads do not own the semaphores as they do with mutexes). Instead this is a signaling system that can be invoked by any threads on any threads. The following declaration and functions are used in the kernel to control kthreads via semaphores:

  - **struct semaphore sem** - declare a semaphore
  - **void sema_init(&sem, value)** - initialize the semaphore counter to `value`
  - **int down_interruptible(struct semaphore *sem)** - decrement the semaphore pointed to by `sem`. If the value of that semaphore is 0 the thread invoking this function is blocked. This version of `down` allows a user-space process that is waiting on a semaphore to be interrupted by the user.
  - **void up(struct semaphore *sem)** - increment the semaphore by 1. Any blocked thread will be unblocked by will return the semaphore to 0 as it returned from `up`.

  ▷ **Atomic** - the simplest of the approaches to kernel synchronization, and the easiest to understand and use. Atomic actions operate in one `uninterruptible` operation. The following declaration and functions are used in the kernel to control kthreads via semaphores:

  - **atomic_t val** - declare an atomic signed integer
  - **atomic_set(&val, v)** - set value of `val` to `v`
  - **atomic_add(v, &val)** - add `v` to `val`
  - **atomic_dec(&val)** - decrement `val`
  - **atomic_read(&val)** - return the value of `val`

- Locks

  ▷ **Mutex** - may be used in critical sections that go to sleep, unlike the spinlock. A thread takes ownership of a mutex lock and other threads cannot unlock it although they can also take the lock if the original mutex owner is sleeping in the critical section the lock protects. Declarations and operations follow:

  - **struct mutex lock** - declare a mutex lock
  - **mutex_init(&lock)** - initialize the lock
  - **mutex_lock(&lock)** - get the lock
  - **mutex_unlock(&lock)** - release the lock

  ▷ **Spinlock** - Most kernel locking is implemented with spinlocks. Unlike semaphores, spinlocks must be used in code that cannot sleep (e.g. interrupt handlers). Spinlocks can offer higher performance than semaphores in general, but are subject to a different set of constraints.
  constraint 1: the critical section holding the spinlock must be atomic. It cannot sleep. It must be guaranteed to reach the unlock. otherwise some other piece of code that takes the processor while the critical section is sleeping may try to get the lock and it cannot do that.
  constraint 2: When the kernel holds a spinlock, preemption is disabled!
  constraint 3: The locked code must finish as fast as it possibly can! Otherwise, perhaps even the CPU scheduler is in trouble!
  The following are the declaration and operations:

  - **spinlock_t lock** - declare a spin lock
  - **spin_lock_init(&lock)** - initialize the lock
  - **spin_lock_irqsave(&lock, flags)** - get the lock
  - **spin_unlock_irqrestore(&lock, flags)** - release the lock

- **Work queue**

  Used to manage work deferral. Processing of work may need to be deferred when an interrupt occurs. Typically, some work needs to be done in the context of the interrupt and some work needs to be passed to other OS elements for added processing. In interrupt context latency may be high because some interrupts may be disabled then. So minimizing work in the interrupt context is desired. This entails pushing some work into the kernel context where interrupts are all enabled - thus performance is not affected.

  ▷ **Top half:** an interrupt handler is invoked when an interrupt occurs. It is typically divided into two separate parts. The top half is the part that is executed first, with interrupts turned off. The top half must do a minimum amount of work and exit quickly. Primarily, it sets up and schedules the real work that has to be done (the bottom half) and uses a work queue to defer that work so it can be done in kernel context. The top half is needed, for example, to record platform-specific critical information which is only available at the time of the interrupt.

  ▷ **Bottom half:** completes long interrupt-processing tasks that are prepared by the top half. A bottom half has either a dedicated kernel thread, or is executed from a pool of kernel worker threads. These threads sit on a work queue until processor time is available for them to perform processing for the interrupt. Top halves may have a long-lived execution time, and thus are typically scheduled similarly to threads and processes.

Inter-process communication

- **Producer-consumer**

  A process may need to communicate an exception or some request for service to a responder without having to stop or suspend execution. The responder should be listening for such communication and act when input is available. Co-routines allow this.

- **Signals**

  A system message sent from one process to another, not usually used to store information but instead give commands. Example: `kthread_stop` sends a signal to `kthread_should_stop` to break out of a loop to stop a thread.

- **Procfs**

  A special filesystem that presents information about processes and other system information in a file-like structure, providing a convenient and standardized method for dynamically accessing process data held in the kernel.

- **Sysfs**

  Exports information about devices and drivers from the kernel device model to user space, and is also used for configuration.

Contexts

- **kernel context**

  The memory (registers etc.) of what the kernel was doing on behalf of a user process (namely, servicing system calls) which must be saved when a user process enters a wait state. This frees the kernel to continue working on some other process. An interrupt is raised to awaken the user process, the kernel restores the context and continues handling the user process. The kernel context typically contains holds information about devices, queues, and process control blocks: a kernel data structure containing the information needed to manage a particular process.

- **interrupt context**

  The memory of what the kernel was doing when servicing an interrupt. When an interrupt is handled by the kernel, the kernel is not preemptible. In kernel context the kernel is preemptible.

- **user context**

  The memory of what the user was doing when the user process is suspended, say because its time slice is up.

Scheduling and Dispatching Processes

Motivation

- An OS intends to support far more processes than there are processors in hardware. Some time should probably be given to each active process but typically a system runs better if the decision to preempt a running process and dispatching a suspending process in its place is allowed to vary, depending on the type of process and other factors. This is why there are so many scheduling algorithms

Performance Criteria

- Criteria for measuring how well a scheduling algorithm performs are conflicting. Examples are as follows:
  - ▷ **Throughput:** # procs completed per unit time
  - ▷ **Latency:** Time waiting for process to start
  - ▷ **Response time:** Time from dispatch to termination
  - ▷ **Fairness:** % of cpu cycles given to a process
  - ▷ **Priority:** Processes may be more urgently granted cpu cycles
  - ▷ **Starvation:** Process is stuck because it cannot get resources needed to finish
  - ▷ **Overhead:** Time to switch context for a change in process serviced by a CPU

Scheduling Algorithms

- **FIFO:** Processes are placed in a single queue and run to completion when they are taken off the queue.
  - ▷ **Throughput:** bad - some processes may hog time
  - ▷ **Fairness:** No - in a time period a process gets many or few cpu cycles
  - ▷ **Priorities:** None
  - ▷ **Starvation:** Possible if never-ending process is enqueued
  - ▷ **Overhead:** Very low
  - ▷ **Latency:** Unpredictable
  - ▷ **Response time:** Unpredictable
  - ▷ **Summary:** Forget about a single FIFO for scheduling
- **Round Robin:** Processes are given a *slice* of CPU cycles then suspended and resumed later. The order in which processes are given time is fixed, usually depends on when the *ready* queue is entered.
  - ▷ **Throughput:** Varied - low for lots of jobs of low duration
  - ▷ **Fairness:** Very - every job gets equal cycles (unless priorities are added)
  - ▷ **Priorities:** None in pure form, can be layered into this scheme
  - ▷ **Starvation:** No - every job gets cpu cycles until finished
  - ▷ **Overhead:** high - many context switches per job
  - ▷ **Latency:** Low - all jobs start soon after being enqueued
  - ▷ **Response time:** High - many jobs can share CPU
  - ▷ **Summary:** Pretty decent general purpose scheduling algorithm

- **Shortest Running Time:** The job with the least estimated remaining processing time is selected next. This can be managed by a priority queue but the insertion complexity is $\log(n)$ instead of $O(1)$ for a normal queue.
  - ▷ **Throughput:** high - short jobs finish fast, skew statistics
  - ▷ **Fairness:** No
  - ▷ **Priorities:** None in pure form, can be layered into this scheme
  - ▷ **Starvation:** Possible - lots of short jobs may prevent a long one from running
  - ▷ **Overhead:** varied - short jobs may preempt longer ones
  - ▷ **Latency:** Unpredictable
  - ▷ **Response time:** Unpredictable
  - ▷ **Summary:** decent results are possible
- **Fixed Priority Preemptive:** Each job has a priority rank, jobs are selected from a priority queue. Entering high priority processes interrupt lower priority processes.
  - ▷ **Throughput:** Could be bad - some processes may hog time
  - ▷ **Fairness:** No unless it is fair that high priority trumps low priority
  - ▷ **Priorities:** Yes
  - ▷ **Starvation:** Possible - high priority jobs may prevent low priority jobs from getting CPU cycles
  - ▷ **Overhead:** Moderate - $O(\log(n))$ time to decide
  - ▷ **Latency:** Unpredictable
  - ▷ **Response time:** Unpredictable
  - ▷ **Summary:** Not one of the best choices by itself
- **Completely Fair Scheduling:** Select the job with the least spent processing time. Managed by a red-black tree with $O(\log(n))$ insertion times.
  - ▷ **Throughput:** Not bad - cycle hogs eventually get fewer CPU cycles
  - ▷ **Fairness:** Yes - a job that spends a lot of time sleeping does not get penalized for that and when it awakens it will likely be given a shot of processing time since its spent time is low. Interactive processes get handled quickly at the same time CPU bound processes hardly know that they have lost a few time slices.
  - ▷ **Priorities:** No
  - ▷ **Starvation:** Possible - long jobs may get fewer cycles the longer they stay alive
  - ▷ **Overhead:** Moderate - $O(\log(n))$ time to decide
  - ▷ **Latency:** Unpredictable
  - ▷ **Response time:** Good - see fairness
  - ▷ **Summary:** Very good choice for general purpose

- **BFS:** Extremely simplified version of CFS
  - ▷ **Throughput:** Not bad
  - ▷ **Fairness:** Yes
  - ▷ **Priorities:** None in pure form, can be layered into this scheme
  - ▷ **Starvation:** Possible
  - ▷ **Overhead:** Low
  - ▷ **Latency:** Very low
  - ▷ **Response time:** Good
  - ▷ **Summary:** Good for small machines
- **Fixed Priority Preemptive:** Each job has a priority rank, jobs are selected from a priority queue. Entering high priority processes interrupt lower priority processes.
  - ▷ **Throughput:** Could be bad - some processes may hog time
  - ▷ **Fairness:** No unless it is fair that high priority trumps low priority
  - ▷ **Priorities:** Yes
  - ▷ **Starvation:** Possible - high priority jobs may prevent low priority jobs from getting CPU cycles
  - ▷ **Overhead:** Moderate - $O(\log(n))$ time to decide
  - ▷ **Latency:** Unpredictable
  - ▷ **Response time:** Unpredictable
  - ▷ **Summary:** Not one of the best choices by itself
- **Multi-level Feedback Queue:** Lots of process-ready FIFOs with level numbers 1,2,...N-1 and a Round Robin queue at level N. Processes at lower level numbers have higher priority. A process entering the MFQ is placed at the end of the lowest level FIFO. When it runs, if it does not complete after a certain time interval it is placed at the end of the FIFO at the next higher level. This keeps happening until it completes or enters the RR queue where it will stay until it finishes. A process waiting too long in a higher level may be moved to a lower level.
  - ▷ **Throughput:** Not bad
  - ▷ **Fairness:** Some - gives preference to short and I/O bound processes
  - ▷ **Priorities:** Yes
  - ▷ **Starvation:** No
  - ▷ **Overhead:** Moderate - but can effectively use multiple cores
  - ▷ **Latency:** Varied
  - ▷ **Response time:** Varied
  - ▷ **Summary:** Pretty good choice - used by Windows

Memory Barriers

Motivation

- Reordering memory references allows much better performance, and so memory barriers are needed to force ordering in things like synchronization primitives whose correct operation depends on ordered memory references. Memory barriers are a necessary evil that is required to enable good performance and scalability, an evil that stems from the fact that CPUs are orders of magnitude faster than are both the interconnects between them and the memory they are attempting to access.

Cache structure

- A CPU can execute many instructions per nanosecond
- A memory fetch takes 60 nanoseconds
- Recent memory fetches are copied to a cache whose speed is is similar to that of a CPU - the CPU fetches from the cache first (a hit) but if the reference has not been copied (a miss) the fetch from memory must occur (takes a long time). The copies are typically done a line at a time (2^4 to 2^8 bytes) and such blocks are called "cache lines."
- There is one cache per CPU
- If the cache is full, some cache line is going to have to be evicted to allow room for the requested memory copy.
- The cache is organized as a hash table so finding a cache line to address is pretty fast. But this means evictions may take place even if the cache is not full. The cache line address is usually just 4 bits of the memory address (hash table size is then 16 entries times the number of "ways").
- Several CPUs may have the same copy of memory data.
- If there is a write to memory, the write is made to the cache. Since all CPUs must agree on the value of a given data item before a given CPU writes that data item, it must first be removed, or "invalidated," from other CPUs' caches. Once completed, the CPU may safely modify the data item. If the data item was present in this CPU's cache, but was read-only, this process is termed a "write miss." Once a given CPU has completed invalidating a given data item from other CPUs' caches, that CPU may repeatedly write (and read) that data item.
- Much care must be taken to ensure that all CPUs maintain a coherent view of the data. With all this fetching, invalidating, and writing, it is easy to imagine data being lost or (perhaps worse) different CPUs having conflicting values for the same data item in their respective caches. These problems are prevented by "cache-coherency protocols."

Cache-coherency protocol

- Cache-coherency: assurance that changes in the values of data items shared among caches are propagated throughout the system in a timely fashion. Three levels of cache coherence are:

    1. Every write operation appears to occur instantaneously
    2. All processors see exactly the same sequence of changes of values for each separate operand
    3. Different processors may see an operation and assume different sequences of values

- Cache-coherency protocols manage cache-line states so as to prevent inconsistent or lost data.
- This entails sending messages between CPUs that cause cache-lines to be marked invalid in case a change has been made to a duplicated data-line, or modified if the changes are local to a single CPU, exclusive if the cache-line is up-to-date, or shared meaning other CPUs have this cache-line and it cannot be updated without consent from other CPUs.
- Reordering memory references allows much better performance, and so memory barriers are needed to force ordering in things like synchronization primitives whose correct operation depends on ordered memory references. Memory barriers are a necessary evil that is required to enable good performance and scalability, an evil that stems from the fact that CPUs are orders of magnitude faster than are both the interconnects between them and the memory they are attempting to access.

Processor guarantees the following

- On any given CPU, dependent memory accesses will be issued in order, with respect to itself. For example, the statements `Q = P; D = *Q;` will be executed in that order, not reversed.
- Overlapping loads and stores within a particular CPU will appear to be ordered within that CPU. For example, `a = *X; *X = b;` and `*X = c; d = *X;` will be executed in that order.

Assumptions:

- It must be assumed that independent loads and stores can be issued in an order different from the given order. For example, `X = *A; Y = *B; *D = Z;` may be executed in the order `Y = *B; *D = Z; X = *A;`.
- It must be assumed that overlapping memory accesses may be merged or discarded. For example,

```
static inline void incr_bp(volatile unsigned long *index, int d) {
    unsigned long new = *index + d;
    *index = (new >= (buffer + PAGE_SIZE)) ? buffer :  new;
}
```

  may be executed as

```
static inline void incr_bp(volatile unsigned long *index, int d) {
    *idx = ((*idx+d) >= (buffer + PAGE_SIZE)) ? buffer : (*idx+d);
}
```

Barrier types

- **Write:** gives a guarantee that all the STORE operations specified before the barrier will appear to happen before all the STORE operations specified after the barrier with respect to the other components of the system.
- **Read:** gives a guarantee that all the LOAD operations specified before the barrier will appear to happen before all the LOAD operations specified after the barrier with respect to the other components of the system.
- **Data dependency:** In the case where two loads are performed such that the first load retrieves the address to which the second load will be directed, a data dependency barrier makes sure that the target of the second load is updated before the address obtained by the first load is accessed.

Example:

```
int A = 1;
int B = 2;
int C = 3;
int *P = &A;
int *Q = &C;

B = 4;   <- must ensure this is set before proceeding so put in a
wmb()       write memory barrier here
P = &B;  <- this and next statement stay in order by cpu rules
Q = P;   <- correct if P has been changed to &B and B = 4
rmb()    <- put a read barrier here to make sure effects of middle
            lines is visible to last line
D = *Q;  <- correct if Q is the same as P and B = 4 - if this is
            executed on another cpu, the wmb() earlier makes sure
            the value of *Q is 4
```

Microcode

Motivation

- Low-level code that defines microprocessor operations For each machine-language instructions
- One machine-language instruction translates into several microcode instructions
- Microcode may be stored in ROM (cannot be modified) or stored in EPROM (can be updated).

Detailed example

- See lecture notes on microcode

Architecture

- See lecture notes on microcode

Horizontal vs. Vertical

- See lecture notes on microcode

IO Channels

Motivation

- I/O tasks can be complex
- I/O tasks can be a lot slower than the data bus
- The CPU could waste time handling I/O
- I/O Channel processors are simple
- Communication with the CPU is via interrupts. The OS sets up a program for delivery to the channel.

What's Wrong With This Code?

Protect the test!

```
#define LOOPS 1000000

int stat[LOOPS] = { 0 };
int list[LOOPS] = { 0 };
int idx = 0, cs1 = 0, cs2 = 0;
struct task_struct *t1, *t2;
struct semaphore lock;

int consumer(void *ptr) {
   printk(KERN_INFO "Consumer TID %d\n", (int)ptr);

   while (idx < LOOPS) {
      if (!down_interruptible(&lock)) {
         list[idx++] += 1;
         if ((int)ptr == 1) cs1++; else cs2++;
         up(&lock);
      }
   }
   printk(KERN_INFO "Cons %d done, cs1=%d cs2=%d\n",(int)ptr,cs1,cs2);
   return 0;
}

int init_module (void) {
   sema_init(&lock, 1);

   t1 = kthread_run(consumer, (void*)1, "cons1");
   t2 = kthread_run(consumer, (void*)2, "cons2");
   return 0;
}

void cleanup_module(void) {
   int i;
   for (i=0 ; i < LOOPS ; i++) stat[list[i]]++;
   for (i=1 ; i < 5 ; i++)
      if (stat[i] > 0) printk(KERN_INFO "%d: %d",i,stat[i]);
   printk(KERN_INFO "threads: 1=%d 2=%d  # incrs: %d",cs1,cs2,cs1+cs2);
   printk(KERN_INFO "completed\n");
   printk(KERN_INFO "BTW - list[%d] is %d\n",LOOPS,*((&list[LOOPS-1])+1));
}
```

What's Wrong With This Code?

Possible Wrong Return Value for get!

```
typedef struct {
   pthread_mutex_t lock;
   pthreadx_cond_t notifier;
   void *value;      /* object to be passed from consumer to producer */
   bool available;   /* 1 if there is an unread object to pass, otherwise 0 */
} Stream;

void *get(void *stream) {
   pthread_mutex_lock(&((Stream*)stream)->lock);
   if (!((Stream*)stream)->available)
      pthread_cond_wait(&((Stream*)stream)->notifier, &((Stream*)stream)->lock);
   ((Stream*)stream)->available = false;
   pthread_cond_signal(notifier);
   pthread_mutex_unlock(&((Stream*)stream)->lock);
   return ((Stream*)stream)->value;
}

void put(void *stream, void *value) {
   pthread_mutex_lock(&((Stream*)stream)->lock);
   if (((Stream*)stream)->available)
      pthread_cond_wait(&((Stream*)stream)->notifier, &((Stream*)stream)->lock);
   ((Stream*)stream)->available = true;
   ((Stream*)stream)->value = value;
   pthread_cond_signal(notifier);
   pthread_mutex_unlock(&((Stream*)stream)->lock);
}

void *producer (void *stream) {
   int i; for (i=1 ; ; i++)  put(stream, (void*)i);
   pthread_exit(NULL);
}

void *consumer (void *stream) {
   int i; for (i=0; i < 10; i++) printf("got %d\n", (int)get(stream));
   pthread_exit(NULL);
}

void init_stream (Stream *stream) {
  /* available = false, value = NULL, lock = unlocked */
}

int main () {
   pthread_t prod, cons; Stream stream; init_stream(&stream);
   pthread_create(&prod, NULL, producer, (void*)&stream);
   pthread_create(&cons, NULL, consumer, (void*)&stream);
   pthread_join(cons, NULL);
   pthread_cancel(prod);
}
```

What's Wrong With This Code?

acpi calls raise interrupt but interrupts are turned off while running get_batt_stat!

```
struct task_struct *ts;

int get_batt_stat(void *data) {
   struct acpi_buffer buffer = { ACPI_ALLOCATE_BUFFER, NULL };
   acpi_status status;
   acpi_handle handle;
   union acpi_object *result;
   int chrg_dischrg, charge;

   printk(KERN_EMERG "Entering loop\n");
   while (!kthread_should_stop()) {
      msleep(1000);
      status = acpi_get_handle(NULL, (acpi_string)"\\_SB_.BAT0", &handle);
      status = acpi_evaluate_object(handle, "_BST", NULL, &buffer);
      result = buffer.pointer;
      if (result) {
         chrg_dischrg = result->package.elements[0].integer.value;
         charge = result->package.elements[2].integer.value;
         printk(KERN_EMERG "discharging=%d charge remaining=%d\n",
                           chrg_dischrg, charge);
        kfree(result);
      }
   }
   return 0;
}

int init_module () {
   ts = kthread_run(get_batt_stat, NULL, "huh");
   return 0;
}

void cleanup_module () {  kthread_stop(ts);  }
```