```
* async queuing system

HEAD: queue (elements), assume linked list (FIFO)

queue: A -> B -> C -> D ... -> N (tail)
the queue is the list of work to be done

Head: submit/add new jobs (producers)
Tail: extract/remove jobs and process them (consumers)

- producers add work to head of queue, then go off do something else.
- consumers pick work items from the tail of the queue, and process it to
  completion.

An sync queue makes sense when time(produce) is much shorter than
time(consume). e.g., user process submits a file to be compressed.

You can have many producers and many consumers.

How to return success/errors back to producer from consumer?
- producer has to stay running
- producer has to submit some "callback" structure or function
- queue system records the callback in the itemlist (and waitq's)
- when consumer is done, it calls the callback on the task, or sends a
  signal(2), etc.
- process can setup a signal handler, or poll(2) on some shared mem to get
results.

TBD: prod/consume rates and queue's eventual states.
- how many consumers?

////////////////////////////////////////////////////////////////////
// producer-consumer queue processing

// d-s for queue
struct queue {
  spinlock_t l; // protects whole itemlist (other d/s could get away with locking subset)
                // also protects 'len'.  If kmalloc and can block inside CS,
                // then use mutex or some other blocking lock.
  list_head itemlist; // actual queue (can be ANY other d-s)
  u_int len, max;     // current length and max length of queue (max may be configurable)
  wait_q producers_wq;// waitq for producers
  wait_q consumers_wq;// waitq for consumers
};

// invoked by callers, such as user processes
int produce(item)
{
  int err;
  // kmalloc: can still stay with spinlock, but need to kfree if len>=max
  lock(l);
  if (len < max) {
    insert_into_head(itemlist, item); // what if this did a kmalloc()?
    len++;
    if (consumers_eq != NULL)
      wakeup(consumers_wq); // wakeup one consumer
  } else {
    //    return -ENOSPC; // bad: leaving 'l' locked
    //    err = -ENOSPC; // harsh response to return err to users
    add_to_waitq(producers_wq, current); // "throttling the heavy writers"
    // may need to have a higher "max" and return real errors.
    if (consumers_eq != NULL)
      wakeup(consumers_wq); // wakeup one consumer (possible to get here)
  }
```

```
    // can move wakeup of consumers here
    unlock(l);
    return err;
  }

  // invoked by kernel threads woken up to "consume" the queue
  int consume(void)
  {
    item_t *item;
    int ret;

    lock(l);
    if (len > 0) {
      item = remove_tail_item(itemlist);
      len--;
      //    if (len < max) // redundant check
      //      wakeup(producers_wq); // wakeup one (maybe more?)
    } else { // len==0
      //    goto out; // bad idea: consumer will cycle b/t READY and RUNNING states
      add_to_waitq(consumers_wq, current); // "throttling" consumers when no work to be done
      //    wakeup(producers_wq); // wakeup one (maybe more?) b/c len==0
    }
    if (producers_eq != NULL)
      wakeup(producers_wq); // wakeup one (maybe more?) b/c len==0
    unlock(l);
    ret = process_item(item); // e.g., compress file (and kfree item)
    // check here if len==0 then go to sleep... but len may not be zero
   out:
    return ret;
  }
```