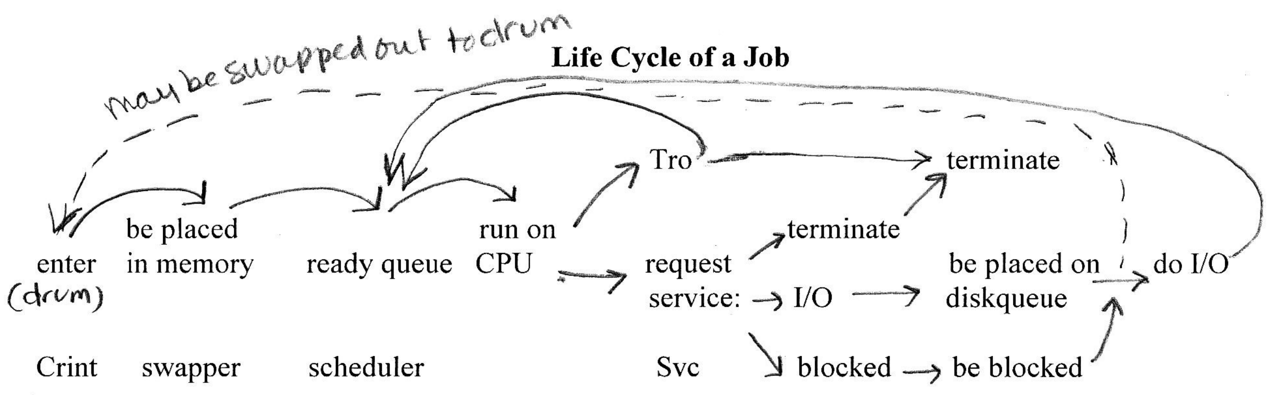
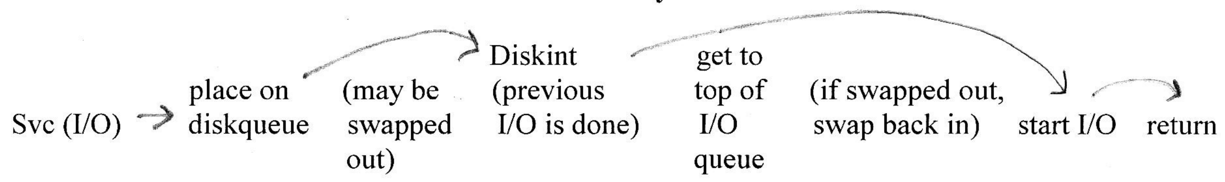
OS Project

The project is a simulation of an operating system. Your program works together with an existing program (SOS) that simulates a job stream. SOS feeds your program a series of jobs and job requests; like a real operating system, your program must field those requests and handle the life cycle of each of the jobs, from system entry to termination.

Life Cycle of a Job



I/O Cycle



LATER:

Diskint (this I/O done) ⟶ start another I/O ⟶ return

Your operating system (OS) does not run real jobs, but simulates interrupt handling, scheduling, memory management, swapping, I/O queueing, job termination, etc. Your OS simulates the software; SOS (my program) simulates the hardware.

The simulated hardware available to your OS is as follows:

1) CPU for running jobs
2) 100K of user memory in 1K blocks, numbered 0-99
3) timer (real-time clock, starts at 0, measures in milliseconds)

Peripherals include:
1) disk for doing I/O
2) drum for swapping
3) input spool to feed in new jobs

My program (SOS) and your program (OS) pass control back and forth through interrupts (SOS passing control to OS), through SIO's (OS starting the hardware), and through a return with instructions to the CPU (OS passing control to SOS). SOS and OS never run at the same time. In the C/C++ version of the program, SOS is the main program which calls OS. SOS first calls a procedure Startup so that OS can initialize its variables.

Here is a diagram of these interactions:

calls issued by OS to SOS functions:	calls issued by SOS to OS functions:
siodisk()	Startup
siodrum()	Crint
ontrace()	Svc
offtrace()	Tro
	Diskint
	Drumint

The system is simplified over a real system: there is no program counter, no registers, no drum management, no disk management. You do not need to allocate space on these devices. Furthermore, the real-time clock has a new value when OS takes over from SOS, but the clock does not change time while your OS is processing.

You will do the following kinds of OS processing: CPU scheduling (long term and short term--you choose the algorithm), memory management (you must use MVP without compaction), swapping, and interrupt handling.

You must write interrupt handling routines to allow OS to take the necessary action when it is interrupted by the users or by the hardware. This means that you must have 5 interrupt handlers (Crint, Tro, Diskint, Drumint, and Svc). Remember that an operating system sits around (sleeps) until something happens that requires its attention. Each function follows this pattern:

```
                              sleep
                                |
                       bookkeep running job
                                |
         take care of the event (Crint, Diskint, Drumint, Tro, Svc)
                                |
   see if there is a process to move into (or out of) memory (Swapper)
                                |
              decide who to run next (Scheduler)
                                |
                          load and run
                                |
                              sleep
```

The part that will differ from routine to routine is "take care of the event." Each routine is only a few lines long, since each calls other routines to do the actual processing.

Each time, except the very first, that an interrupt occurs, a running job will be interrupted. That job will no longer be running on the CPU, and thus the CPU will be idle. It is, therefore, the job of OS to make sure that information about the job that was running is not lost. This task, called Bookkeeping, must be done at the beginning of each interrupt handler. It is also the job of OS to make sure that the CPU is never idle when there is a job in memory that can be run; therefore, OS must schedule a job (if there is one) to run on the CPU before returning control to SOS.

Here is a brief annotation of what the other sheet says about each of the interrupts.

Crint: a new job has arrived. Information about the job is in the p array.

When a Crint occurs, you must save the information about the new job which was passed to OS in p. The information must be stored in a Jobtable–this is a collection of PCB's, one for each job. This may be an array or a linked list of process control blocks. The record for each job will contain all the information about the job that is passed on a Crint, plus any other pieces of information about the status of the job which you must keep track of during processing. All information must be stored in this one place. The Jobtable should be global, so that each routine can access the information without passing information as parameters.

Diskint: The disk has finished an I/O operation. I/O has been finished for job at top of I/O queue.

Drumint: The drum has finished swapping a job in or out of memory. The only parameter is the current time in p[5].

Tro: The running job has run out of time. You must determine whether it has used up its maximum allocation or only a time slice, and act accordingly. The only parameter is the current time in p[5].

Svc: The running job wants service. The parameter *a tells what kind of service: *a=5 means a request to terminate, *a=6 means a request for disk I/O, *a=7 means a request to be blocked. A request to be blocked should occur only after a request for I/O (but not always then). This request indicates that the job wants to be blocked (prevented from running on the CPU) until ALL of its outstanding I/O requests have been completed. When all outstanding I/O has been completed, the job should be unblocked. The second parameter is the current time in p[5].

Here is annotation on the calls which OS makes to activate the hardware (clock, disk, drum, cpu).

siodisk: To start the disk doing an I/O operation, OS calls siodisk() and sends SOS the job number of the job whose I/O is to be done. In C/C++, you may send the job number as a literal or a variable;

```
header: void siodisk(long JobNum);
```

siodrum: To start the drum swapping a job into or out of memory, OS calls siodrum() and sends SOS four parameters: the job number of the job to be swapped, the direction of the swap (drum-to-core or core-to-drum), the size of the job in integer K, and the address of the first block in core (0 to 99).Values can be passed as literals or in variables.

RUNNING A JOB: If there is a job in memory that is not blocked, that job (or one of those jobs) should be run on the CPU when control is returned to SOS from OS. To do this, OS must schedule a job to run prior to returning to SOS. It does this by placing information about the chosen job into certain variables, and then returning to SOS.

If there is a job to run, set *a to 2, and put information about the job in p[2] (memory address), p[3] (job size), and p[4] (time slice). The time slice (in milliseconds) tells the CPU how long it should run the job until issuing a Tro. If there is no job to run, just set *a to 1 and don't put anything into the p array.

Other information:

A job which is currently doing I/O cannot be swapped out. A latch bit or boolean is used to indicate that a job is latched (currently doing I/O). Alternatively, this can be a single variable indicating which job is doing I/O, since only one job will be doing I/O at a time.

When I/O is being done for a job, that job must be in core.

When a job is terminated, it does not have to be swapped out. However, its memory and its jobtable entry must be freed for use by another job.

If a job which is currently doing I/O requests to terminate or is terminated by a Tro, it cannot be killed. OS must wait to kill the job until the I/O operation is completed. The OS should set a kill bit to indicate that this job should be terminated as soon as its current I/O operation is completed.

The swapper is responsible for determining which job that is currently not in memory should be placed in memory (long-term scheduling). It should call a routine that finds space in memory for that job (using first fit or best fit, etc.). Once space is found, OS must call siodrum() to swap the job in. Later on, you may need to swap jobs out to make room for the one that you want to swap in. The swapper will do this job as well.

You must have some representation of memory or free space tables, etc., so that you can find which spaces are free in memory and how large each is. This can be a linked list, or a character string. You must change the values in your own representation of memory as a job is swapped in or out.

You do not need to do any printing except for debugging. SOS does all the printing for you. SOS prints a running account of what is going on. Calls to ontrace() or offtrace() turn on/off the very detailed trace. After you have debugged your program, you should turn the trace off at the beginning of the program. You can turn it back on for debugging later in the program simply by using an **if** statement and testing the value of the timer or the job number at the point at which you want to resume tracing.

Your Jobtable should be declared with room for 50 jobs. Filling the jobtable will cause you to be stopped for processing too slowly.