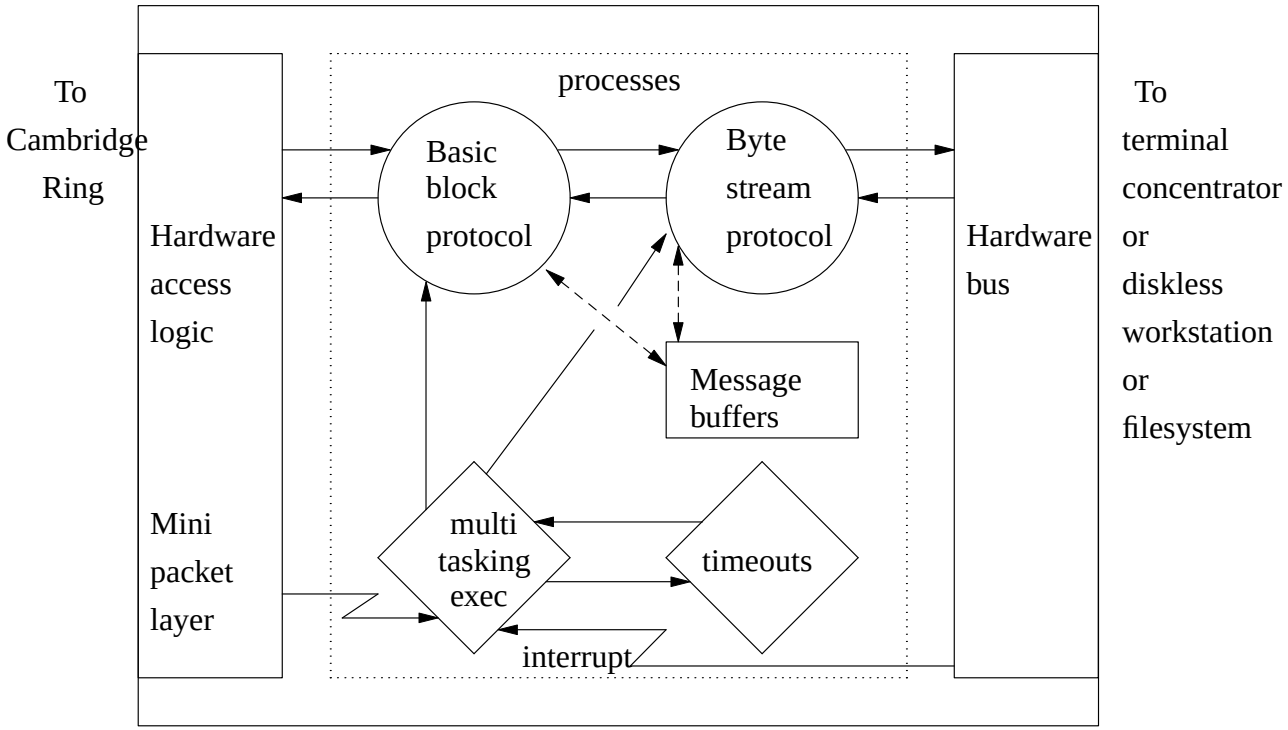
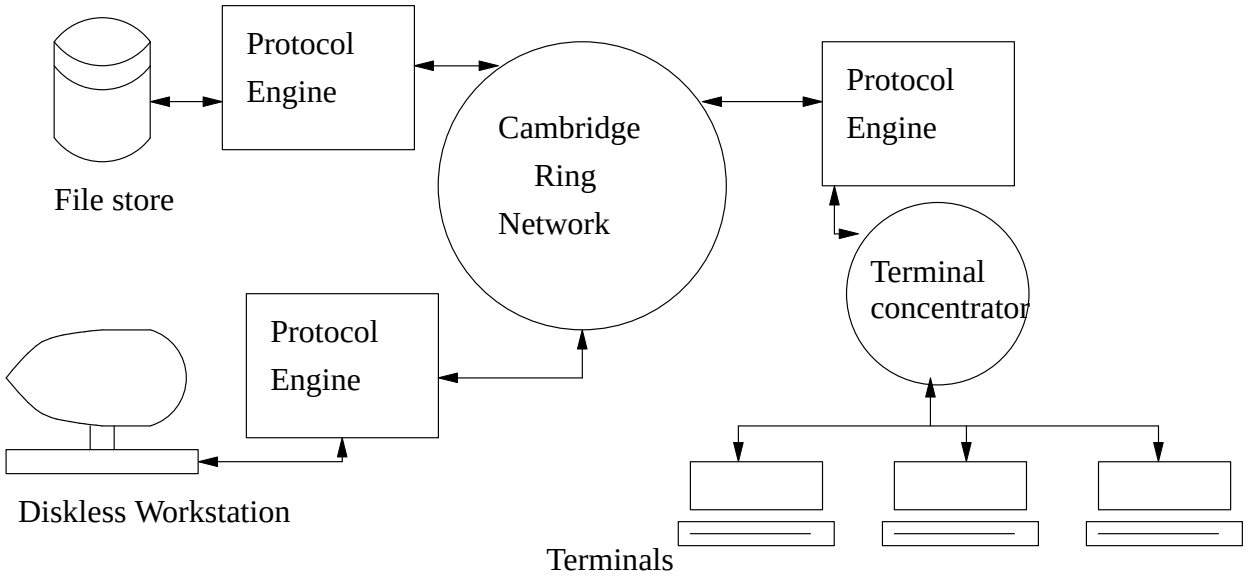
The Executive

- Why do we need an executive?
  - concurrent processes
  - one process, one purpose
  - one process to control one device
  - application separate from device drivers

The Executive



Protocol Engine - example of a microkernel



Protocol Engine - example of a microkernel

- protocol engine must
  - obey the Cambridge Ring protocol
  - send and receive correct data across a network
  - get and put messages must occur concurrently
  - microkernel uses processes coordinated via executive

The Executive

- what does the executive provide?
  - `SEMAPHORE` data type
  - semaphore operations `Wait` and `Signal`

- `InitProcess`
  - to create a new process

The Executive

- `Resume`
  - run the new process

- processes
  - a process can be conceptually thought of as a virtual processor
  - a number of processes running can be thought of as a virtual multiprocessor machine
  - although it is *implemented* on one processor

How do we implement processes

- a process can be a copy of the processor

- a process has its own volatiles
  - own register set: `eax`, `ebx`, `ecx`, `edx`, `ebp`, `esp`, `flags` etc
  - own stack
  - own data
  - own code

How do we implement processes

- it may also share data and code with other processes

- we can give the illusion of running concurrent processes if we:
  - run a slice of one process
  - stop it
  - run a slice of another process
  - etc

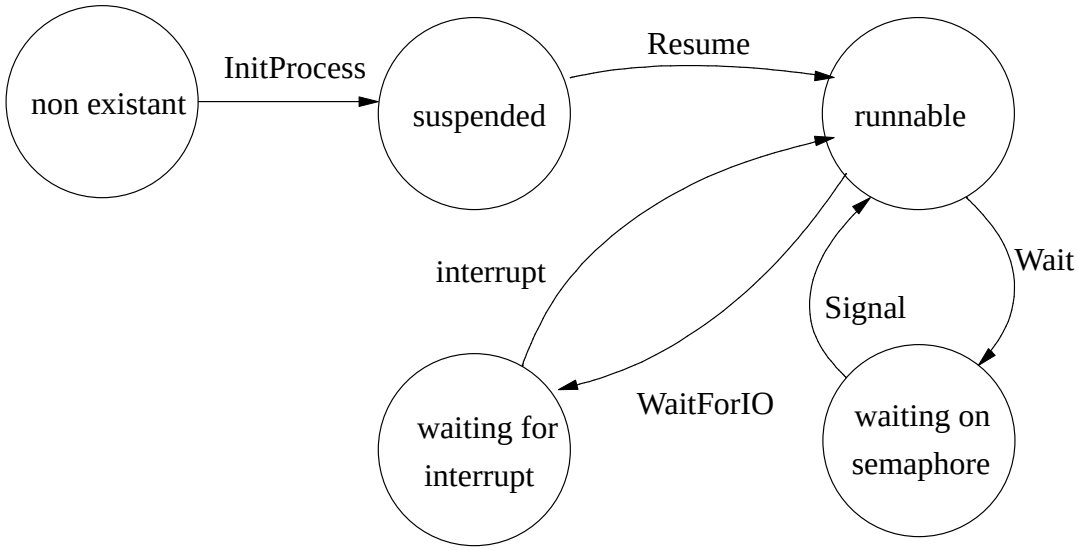
Process states

- note that processes run continually until
  - blocked on a semaphore (`Wait` on semaphore whose value = `0`)
  - blocked waiting for in interrupt to occur `WaitForIO`

- examine the module `Executive.c`

Process states

- we can construct a process state transition diagram:

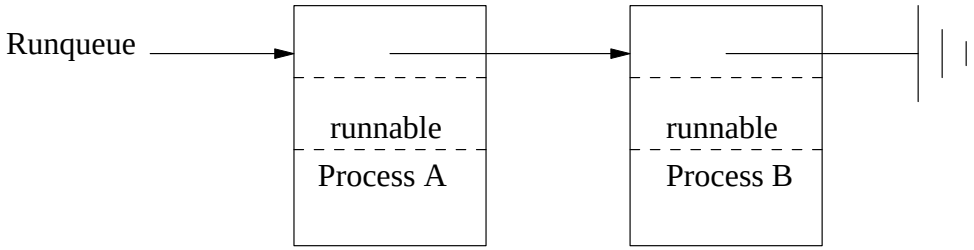


Process states (continued)

- `runnable` means that a process may legally run when the processor has nothing else to do
  - if we have multiple processes then we would sometimes expect to have more than one process in one state. Ie
  - a collection of `runnable` processes
  - a collection of processes waiting on a semaphore

- within the executive it is often useful to collect like entities together
  - typically you would find a run queue which contains all `runnable` processes

Process states (continued)



Process record structure

- a process's record structure should contain all information about that process
  - volatiles (registers)
  - `typedef enum {Runnable, Suspended, WaitOnSem, WaitOnInt} State;`
  - run priority
  - pointers
  - name

- see Executive.c ⟨`src.html#Executive-c`⟩ for details

Implementing Wait and Signal

- consider semaphore operations `Wait` and `Signal`
- remember the pseudo code for `Wait`
- ```
  while (s <= 0)
    ;
  s--;
  ```
- and pseudo code for `Signal`
- ```
  s++
  ```

Implementing Wait and Signal

- could implement semaphores like this but it would be very inefficient, why?

Implementing Wait and Signal

- we note that if semaphore value is <= 0 the process cannot go further
  - therefore we should do something else until `s>0`
  - we could shelve the process and unshelve a process that is able to run. Ie a process which is `runnable`

Implementing Wait and Signal

- the pseudo code of `Wait` now looks like:

```
void Wait (SEMAPHORE s)
{
   save and disable processor interrupts ;
   if value of s > 0
   then
      dec(value of s)
   else
      remove CurrentProcess from run queue ;
      mark CurrentProcess as WaitOnSem ;
      add CurrentProcess to semaphore q ;
      Reschedule
   end
   restore processor interrupts
}
```

Implementation of Signal

- the pseudo code of `Signal` must do the opposite

```
void Signal (SEMAPHORE s)
{
   save and disable processor interrupts ;
   if a process is on s q
   then
      remove process, p, from s q
      alter process, p, status to runnable ;
      add p to the run queue ;
      Reschedule
   else
      inc(value of s)
   end
   restore processor interrupts
}
```

The type `SEMAPHORE`

- the `SEMAPHORE` data structure must have at least two entities
  - a value
  - a queue
- the value is incremented and decremented by `Wait` and `Signal`
- the queue contains a list of processes which are waiting for the value to become > 0

Example: two processes competing for a critical region

```
process A                process B


while (TRUE) {           while (TRUE) {
   Wait(Mutex) ;            Wait(Mutex) ;
   do something             do something
   Signal(Mutex);           Signal(Mutex) ;
}                        }
```

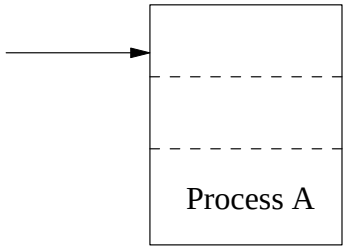
Example: two processes competing for a critical region

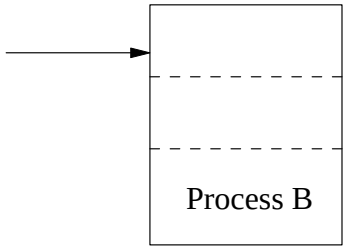
```
SEMAPHORE
        queue
        value



RunQueue
```

1


Process A


Process B

The Executive

- data structures in the executive:
  - queues: `DesQueue`, `SemQueue` which allow various items to be added and removed from queues
  - process `State` which is one of `{Runnable, Suspended, WaitOnSem, WaitOnInt}`
  - indicates which state a process is currently in See process state transition diagram in previous slides
  - `Priority` - run priority of a process
  - A `hi` priority process will always be chosen by the `Reschedule` procedure compared to a `lo` priority process. (You can probably ignore this issue if you wish!)

The executive data structures

- `SEMAPHORE` - describes the state associated with a semaphore
- `DESCRIPTOR` - describes the state associated with a process
- these last two need to be understood in order to complete laboratory exercise executive

SEMAPHORE type

```
typedef struct Semaphore {
  unsigned int       Value;   /* semaphore value                */
  EntityName         SemName; /* semaphore name for debugging   */
  void              *Who;     /* queue of waiting processes     */
  struct SemQueue {
    struct Semaphore *Right,  /* list of existing semaphores    */
                     *Left;
  }                   ExistsQ;
} Semaphore;
```

The type SEMAPHORE

- for laboratory executive you only need to understand 2 fields
  - `Value`
  - `Who`

The type SEMAPHORE

- Value
  - the value of the semaphore
- Who
  - the head of a list of all processes currently waiting on this semaphore (all processes blocked on this semaphore)

The type SEMAPHORE

- `SemName`
  - a `char SemName[MaxName]` which is only used for debugging (`Ps` - will display a list of processes and semaphores)

- `ExistsQ`
  - housekeeping - all semaphores in existence are on a list (so that `Ps` can display each and every semaphore)

- after understanding these fields we should be able to initialize a semaphore (implement `InitSemaphore`)

Implementation of InitSemaphore

```
Semaphore *Executive_InitSemaphore (unsigned int v, char *Name,
                                    const int Name_HIGH)
{
  Semaphore *s;
  OnOrOff ToOldState;

  /* disable interrupts */
  ToOldState = SYSTEM_TurnInterrupts(Off);
  SysStorage_ALLOCATE((void **)&s, sizeof(Semaphore));

  s->Value = v;    /* initial value of semaphore */
  /* save the name for future debugging   */
  StrLib_StrCopy(Name, Name_HIGH,
                 (char *)&s->SemName, MaxCharsInName);

  s->Who = NULL;   /* no one waiting on this semaphore yet */

  AddToSemaphoreExists(s);   /* add semaphore to exists list */

  /* restore interrupts */
  ToOldState = SYSTEM_TurnInterrupts(ToOldState);
  return s;
}
```

The type DESCRIPTOR

- the `DESCRIPTOR` defines a process
  - within the executive we need to keep track of a processes registers, its run state, its name, priority, etc
  - for full details of this data type see `Executive.c` some of these fields are for debugging purposes, others for the scheduler.

- for the purposes of laboratory executive you need to understand the following fields
  - `Volatiles`
  - `ReadyQ`
  - `SemaphoreQ`
  - `Which`
  - `Status`

The type DESCRIPTOR

```
typedef struct Descriptor {
  PROCESS               Volatiles;  /* process volatile environment  */
  struct DesQueue {
    struct Descriptor *Right, *Left;
  }                     ReadyQ,     /* queue of ready processes      */
                        ExistsQ,    /* queue of existing processes   */
                        SemaphoreQ; /* queue of waiting processes    */
  Semaphore            *Which;      /* which semaphore are we waiting*/
  EntityName            RunName;    /* process name for debugging    */
  State                 Status;     /* state of process              */
  Priority              RunPriority;/* runtime priority of process   */
  unsigned int          Size;       /* Maximum stack size            */
  void                 *Start;      /* Stack start                   */
  BOOLEAN               Debugged;   /* Does user want to debug a     */
                                    /* deadlocked process?           */
} Descriptor;
```

Laboratory executive

- in the last tutorial you were a user of the module Executive, this time you are going to implement the following procedures in this module
  - `Resume`
  - `Wait`

- to implement these procedures you must understand
  - the process transition diagram
  - the data structures ⟨27⟩ just mentioned
  - and how a process migrates from one state to another state

Laboratory executive

- the code you have to write is all high level
  - you can call on the services provided in Executive to dequeue and enqueue, processes and semaphores
  - you must correctly manipulate the fields within processes and semaphores

- if you get it wrong - your program will probably crash **big time!**

- moral of the story
  - use the procedures `DebugString` and `Halt`

The queue operations

- to enqueue a process the run queue you need to
  - set its status to `Runnable`
    ie `p->Status = Runnable`
  - add p to the ready queue ie `AddToReady(p) ;`

- to remove a process from the run queue you should
  - remove process, p, from the ready queue ie `SubFromReady(p)`
  - alter its status to an appropriate value
    ie `p->Status = Suspended`

SEMAPHORE queues

- to add a process to a SEMAPHORE queue

```
AddToSemaphore(&s->Who, p) ;
p->Status = WaitOnSem
p->Which = s
```

- to remove a process from a SEMAPHORE queue called, s, and to place onto the ready queue

```
p = SubFromSemaphoreTop(&s->Who) ;
p->Which = NULL ;
p->Status = Runnable ;
AddToReady(p)
```

SEMAPHORE queues

- *note that a process cannot exist on both a semaphore queue and the ready queue*

SEMAPHORE queues

- use the following code to move a process to the top of the run queue (after it has been added to the run queue)

```
RunQueue[d->RunPriority] = d;
```

Pseudo code for `Resume`

```
DESCRIPTOR Resume (DESCRIPTOR d)
{
   disable interrupts ;
   if d is Suspended
   then
      change Status of d to Runnable ;
      add to ready q
      make sure d is at top of
      ready q
      Reschedule
   else
      Halt('not suspended',
            __LINE__, __FILE__)
   end
   restore interrupts ;
   return d;
}
```

Pseudo code for `Suspend`

```
void Suspend ;
{
   disable interrupts ;
   remove process from ready q ;
   alter status ;
   Reschedule ;
   restore interrupts ;
}
```

- the procedure `Reschedule`
  - tests whether there is a process which is at the front of the ready queue
  - if it is not ourself then transfer control to this process