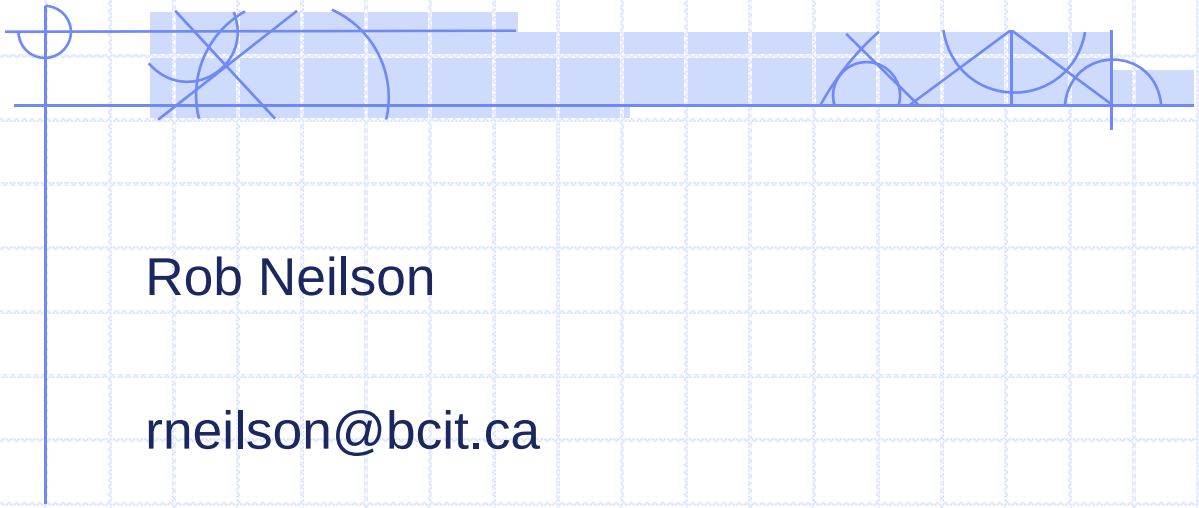
COMP 4735: Operating Systems

Lecture 6: Processes



Rob Neilson

rneilson@bcit.ca

Reading

- The following sections should be read before next Monday
  - it would also help to read this before your lab this week
  - Textbook Sections: 2.1, 2.2 (Theory Part)
    10.3 (Case Study Part)
- Yes, there will be a quiz next Monday in lecture on the above sections
  - A sample quiz will be posted on webct on Friday
  - **This quiz (Quiz 3) covers material from:**
    - **CHAPTER 2.1**
    - **CHAPTERS 10.3.1, 10.3.2, 10.3.3**
    - **THESE LECTURE NOTES**

Agenda

Key concepts for this lesson:

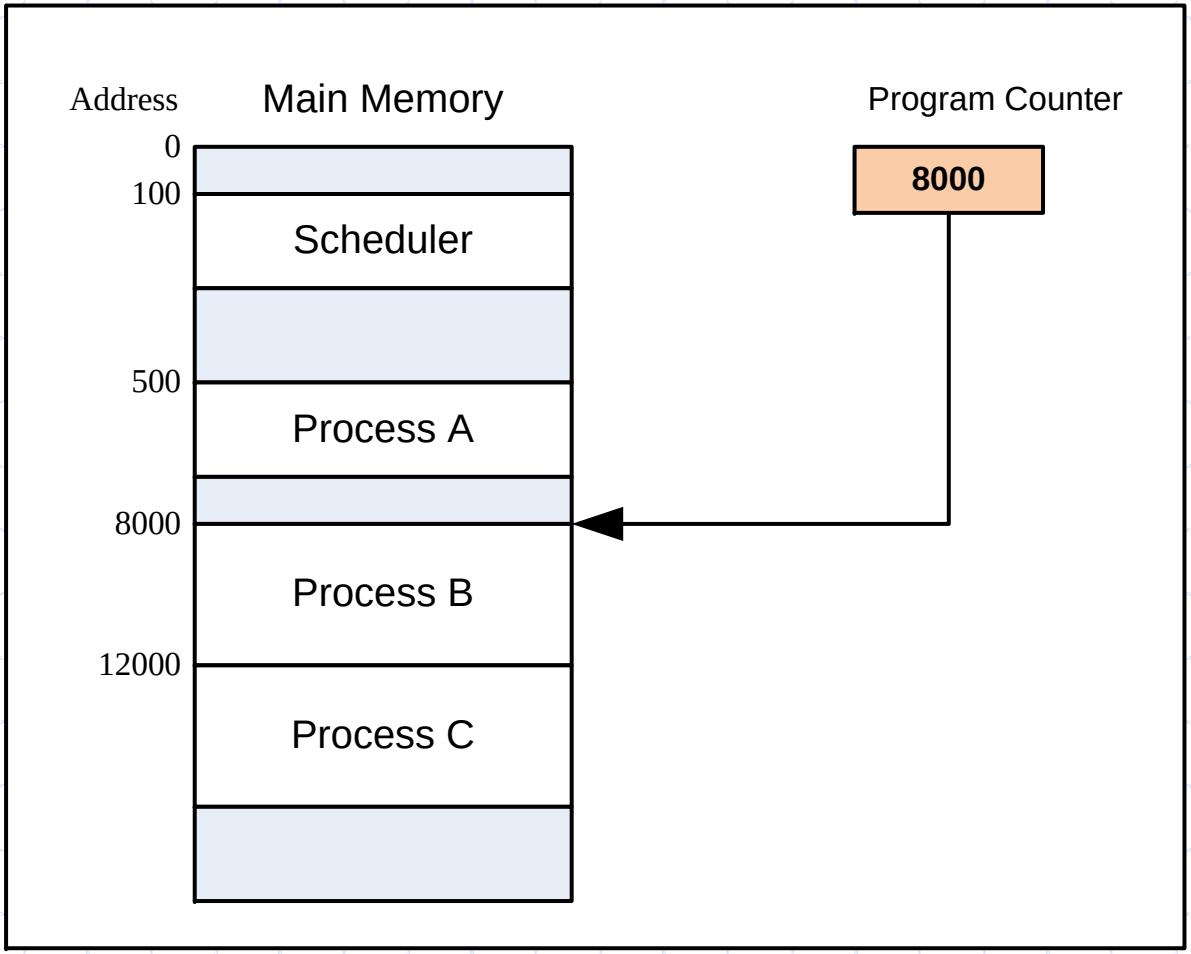
- Multiprogramming
- Process Creation
- Process Termination
- Process States
- Process Table / Process Control Block (revisited)
- Interrupts and ISR's (revisited)

Multiprogramming (review)

- there is only ***one CPU***
- all the programs that wish to execute need to ***share the CPU*** sequentially
- each process (running program) ***uses the CPU for a short period of time*** and then takes a rest so another process can use the CPU
- the act of switching from one running process to another is call ***context switching***
- when a context switch occurs, *all information about the running process needs to be backed up (saved)* so that the process can be restarted where it left off
- the process information is saved in a data structure in the OS called a ***Process Table***, or Process Control Block

Multiprogramming (memory model)

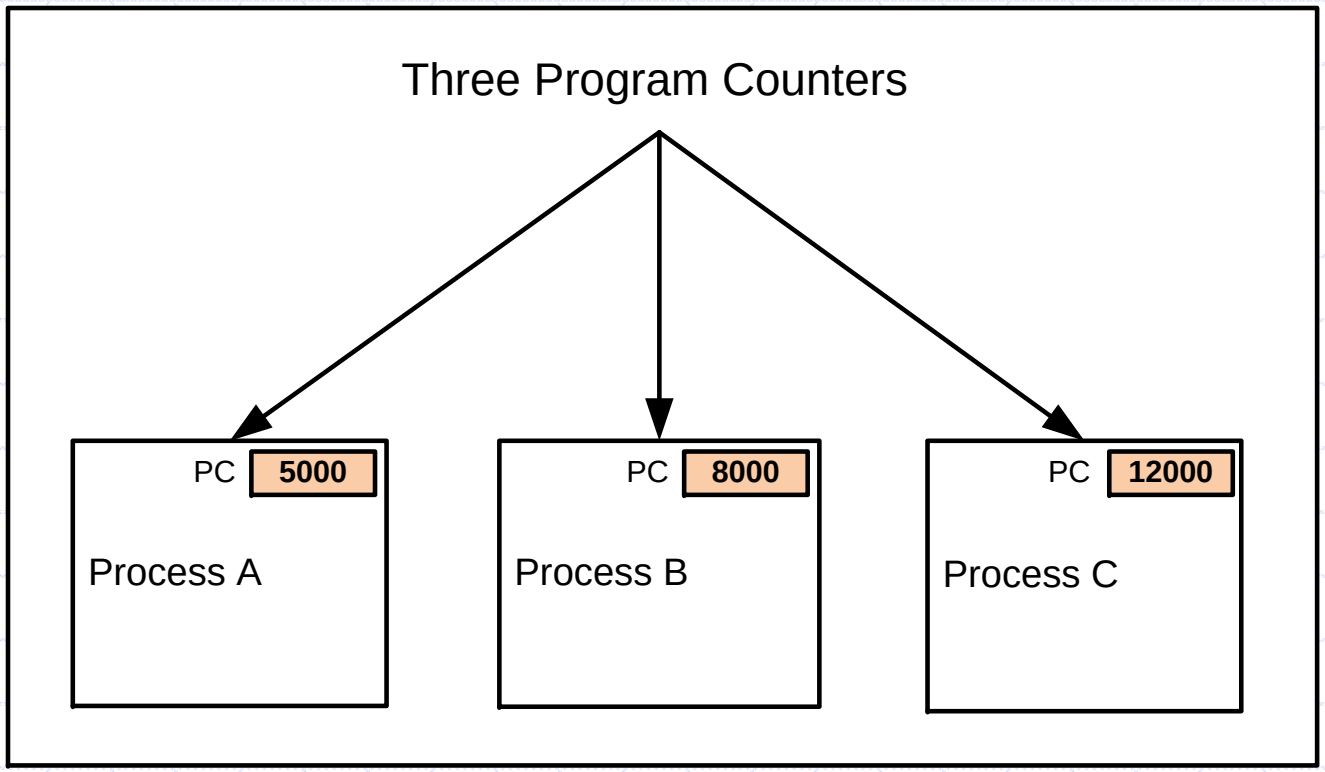
- the following is reproduced from *Stallings: Operating Systems, 5th Ed*



- there is only one CPU
- all programs are loaded into memory
- programs take turns running
- all programs share the program counter

Multiprogramming (process model)

- even though all the processes are loaded into the same RAM, each process thinks it has exclusive access to the computer
- ie: each process has the view that it has an entire CPU and associated registers (eg: program counter) of it's own (but we know this is an illusion)



Multiprogramming (instruction traces)

- each program appears to run as a sequential, uninterrupted process
- instruction traces show the addresses of the program instructions that were referenced when each program executed

(a) Trace of Process A	(b) Trace of Process B	(c) Trace of Process C
5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

5000 - starting address of process A

8000 - starting address of process B

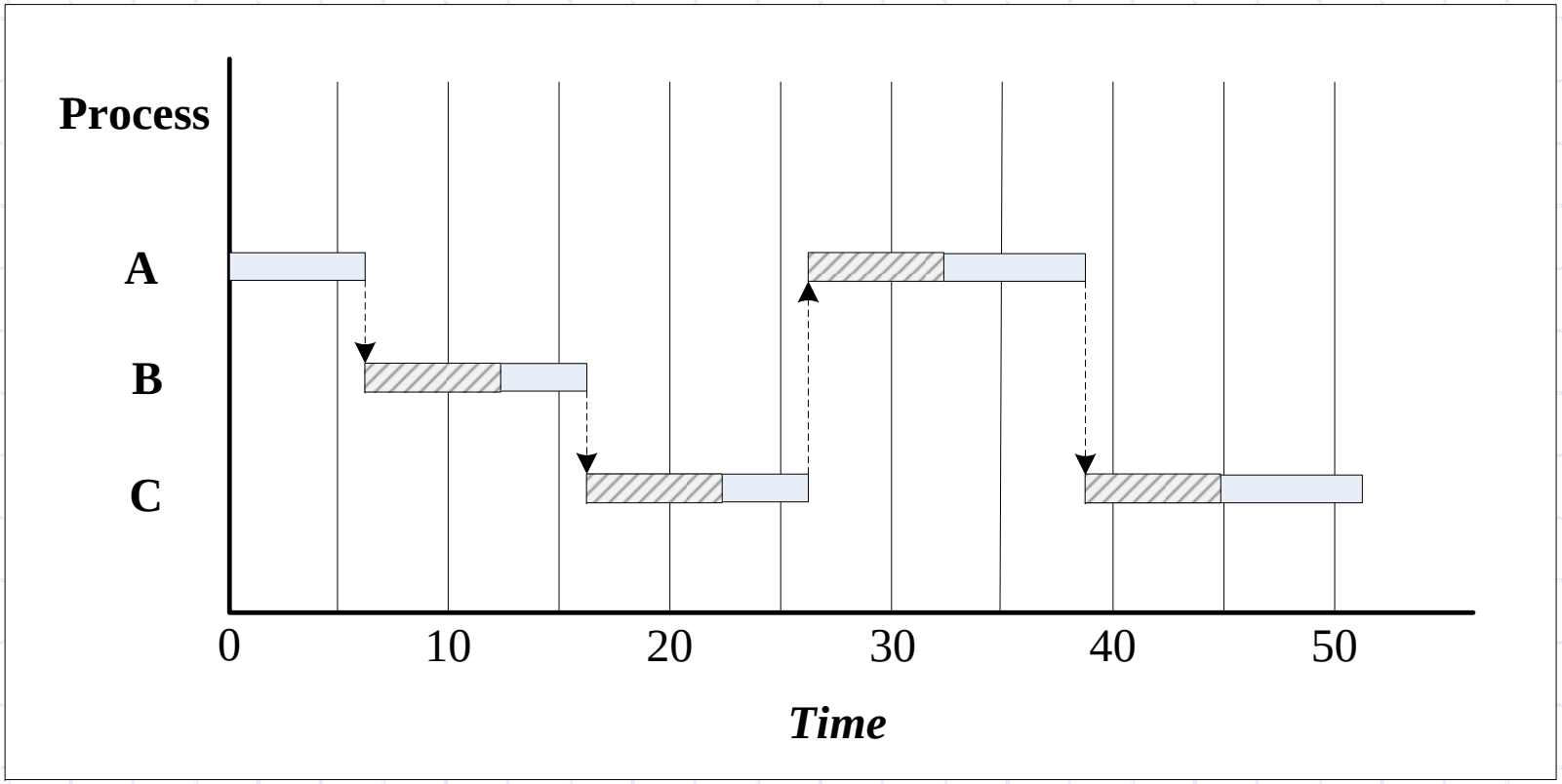
12000 - starting address of process C

Multiprogramming (combined trace)

- the combined trace illustrates what really happens in the CPU when multiprogramming takes place

```
1.   5000
2.   5001
3.   5002
4.   5003
5.   5004
6.   5005
          timeout
7.   100
8.   101
9.   102
10.  103
11.  104
12.  105
13.  8000
14.  8001
15.  8002
16.  8003
          IO Request
17.  100
18.  101
20.  102
21.  103
22.  104
23.  105
24.  12000
25.  12001
26.  12002
27.  12003
28.  12004
29.  12005
          timeout
30.  100
31.  101
32.  102
33.  103
34.  104
35.  105
36.  5006
37.  5007
39.  5008
40.  5009
41.  5010
42.  5011
          timeout
43.  100
44.  101
45.  102
46.  103
47.  104
48.  105
49.  12006
50.  12007
51.  12008
52.  12009
53.  12010
54.  12011
          timeout
```

Multiprogramming (CPU utilization)



- the diagram illustrates how the three processes share the CPU

Multiprogramming Questions

1. Which of the following statements are true about multiprogramming?
   a) In a multiprogramming environment, all processes share a single address space
      - ***no way! each process has its own address space***
   b) Interrupts are needed to initiate context switches in multiprogramming systems
      - ***false ... interrupts can initiate context switching - but so can other events such as the scheduler or a voluntary processor yield by a process***
   c) The state of each process is saved on the stack when the CPU is switches back and forth between processes
      - ***nope :(the state of the process is saved - but in the process table***
   d) There is only one program counter (PC); processes take turns using it
      - ***yes - this is true!! each process thinks it has its own PC - but there is only one; of course each process loads its own addresses into the PC***
   e) There is overhead associated with context switching (multiplexing) processes
      - ***yes there is; the scheduler code that performs the switch needs cpu cycles***

Multiprogramming Questions

2. Which of the following capabilities are enabled by multiprogramming on a single CPU system:
   a) concurrent execution of programs
      - ***sure! this is what multiprogramming does - lets programs run concurrently***
   b) parallel execution of programs
      - ***no way! one cpu means one IR, one PC etc; impossible to have true parallel execution***
   c) concurrent access to physical hardware devices
      - ***sure; concurrent means that processes think they have exclusive use of the device (eg: disk), but it is really shared with other processes***
   d) parallel access to physical hardware devices
      - ***yes; IO operation can happen in parallel as they do not use the cpu; they are enabled by multiprogramming because the associated processes are running concurrently when they start the parallel IO operations***
   e) all of the above are enabled by multiprogramming

Process Creation

There are four types of events that cause processes to be created:

1. At system initialization time
   - very first process is just copied into memory from disk
   - other OS processes are created during the boot sequence
2. By another running process
   - this is the typical mechanism for process creation
   - in unix systems the fork() command creates processes
   - in windows the CreateProcess() command does the work
3. User requests to create processes
   - when users run commands or open windows they are crating processes
4. Batch jobs are started
   - when a batch job is started, a new process is started and it runs the job

Process Creation Questions

3. Which of the following actions ***must*** occur when a process is created on a Unix system?
   a) A process table entry (PCB) is initialized for the process.
      - ***yes***
   b) The child process inherits the parents process identification and status information.
      - ***no - it gets a copy of some status info (not the STATE), but it has its own PID***
   c) The new process loads the program it intends to use.
      - ***no - it doesn't have to load a new program; this is optional (on Unix)***
   d) The parent process allocates the new process a portion of available memory.
      - ***no - any allocations would be done by the OS - not the parent***
   e) None of the above.

Process Creation Questions

4. How is the first process created?
   a) It is copied directly into memory (from disk) by the loader.
      - ***TRUE - this is exactly what happens***
   b) It is created by init when you first log in from a terminal.
      - ***init is a process - so it cannot create the first one***
   c) The BIOS is the first process. It starts running when the system boots.
      - ***the BIOS is not a process, just some instructions and addresses***
   d) The first process is created by the fork command, which is run by the second stage loader.
      - ***what created the process that did the fork??***
   e) None of the above.

Process Table / Process Control Block (revisited)

Process management	Memory management	File management
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Process Table Questions

5. Which of the following items are usually stored in a process table?
   a) List of child processes
      - ***yes, it usually stores this - or a process group ID***
   b) List of file descriptors
      - ***yes, it stores this***
   c) Parent process ID
      - ***yes, it stores this***
   d) State Information
      - ***yes, it stores this***
   e) Working directory
      - ***yes, it stores this too***

Process Table Questions

6. The process table is stored on the stack when the process is not executing on the CPU.

   a) True

   b) False

   – ***the process table is a kernel data structure which is stored in kernel memory; there is no stack involved***

Process Termination (1)

There are four ways a process can be terminated

1. normal exit (voluntary)
   - the *program is finished* and issues the exit command
   - the user closes a window
2. error exit (voluntary)
   - the program is *not finished*, but *needs to terminate* for some reason
     - (eg: incorrect parameter passed on command line)
   - the programmer caused this to happen - probably through error checking

Process Termination (2)

3. fatal error (involuntary)
   - a *fatal run-time error* has occurred, such as running out of memory
   - the OS terminates the process
   - in unix the OS copies the memory image in case a programmer wants to see why the process was killed (ie: a core dump is created)
4. terminated by another process (involuntary)
   - process doing the killing requires authorization
   - Unix sends a signal, which is a form of software interrupt
   - the default action when receiving a signal is to terminate
   - processes can define their own signal handlers to trap signals and take custom actions (something other than terminating)

Signals

- as mentioned, a signal is a form of software interrupt
- you can send a signal to another process with the **kill(2)** system call, or the **kill(1)** shell command
- parameters to **kill** are a pid (process ID) and a signal number
- here are some of the standard signals ...

Signal	*Value*	*Action*	*Comment*
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from *abort*(3)
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGALRM	14	Term	Timer signal from *alarm*(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25		Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process

The signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored.

Signals (2)

- as mentioned previously, a process can *catch* a signal
- to do this (catch a signal) you must write a *signal handler*

- you install (register) your signal handler with the **`signal(2)`** system call

- your signal handler does not need to do anything related to termination
  - for example, you could send a signal to a process when you want to wake it up to do something, like process some information you just placed on a queue ...

- note: a process can only send signals to processes that are in the same *process group*
  - ie: the processes are parents, children, uncles, aunts

Process Termination Questions

7. A process in the running state may involuntarily be removed from the CPU by its parent.

   a) True

   b) False

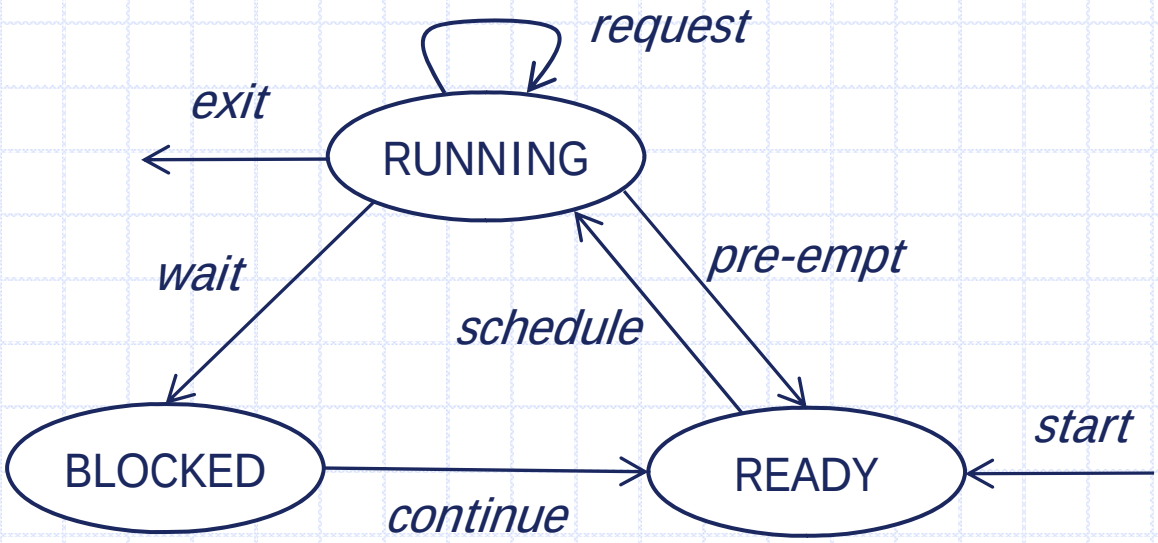
   – ***IMPOSSIBLE - if it is RUNNING the parent does not have the CPU and cannot kill it***

Process Termination Questions

8. Which of the following are true about process termination in Unix systems?
   a) A process can be terminated from the command shell with the kill(1) command.
      - ***yes, especially with kill -9***
   b) A process can terminate itself.
      - ***sure, the exit() call does just this***
   c) A process is able to terminate (kill) a process that is it not related to (ie: a process that is neither an ancestor or a descendant).
      - ***no - it can only kill processes in the same process group***
   d) The default action for many received signals is to terminate.
      - ***yes, this is true***
   e) You can program a process to ignore some termination requests from other processes.
      - ***yes, you can define your own signal handler and catch the signals***
      - ***but you cannot catch SIGKILL or SIGSTOP***

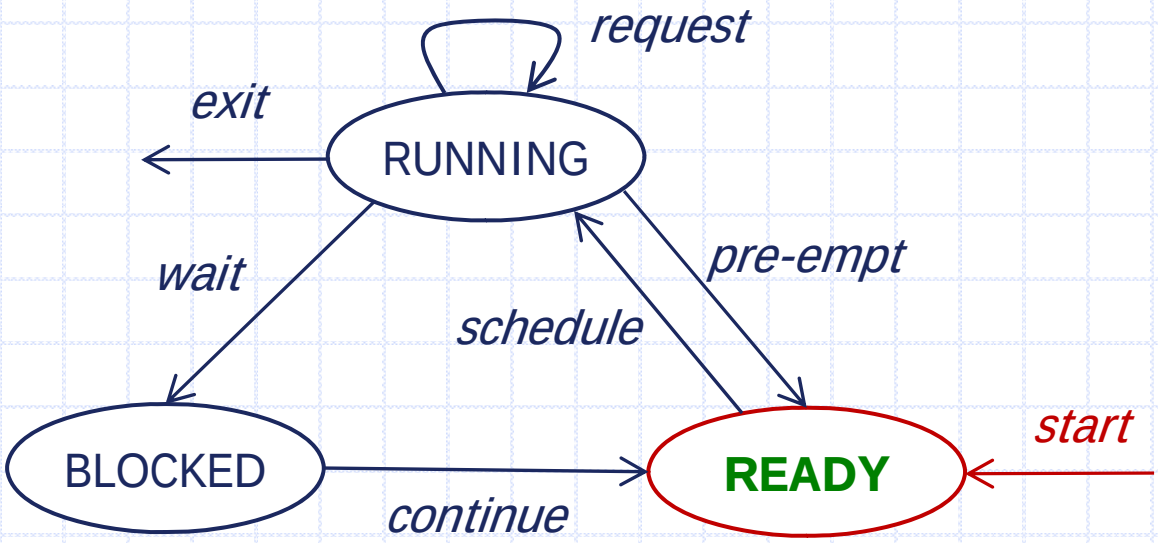
Process States

- All processes are required to be in a *state*.
- We model the possible states for processes using a *state transition diagram.*
  - bubbles are possible states
  - arrows represent events that cause a transition between states
- In the most simple case there will be *exactly three states* that any process can be in, as shown below



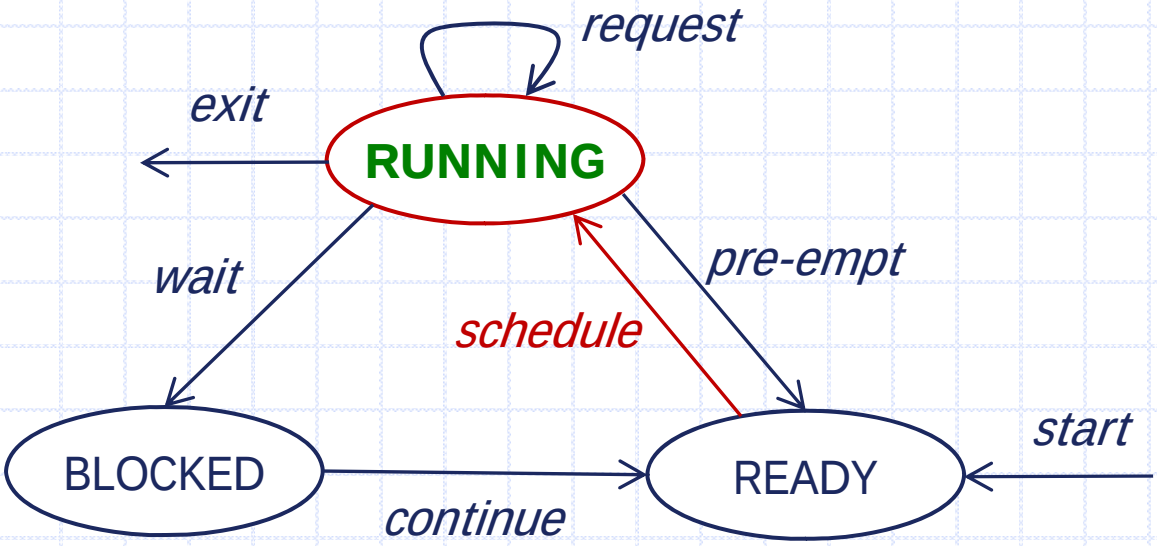
State Transition (1)

- when a new process has been created, a *start* event occurs
- the processes state is set to **READY**
- it is placed in a queue that is managed by the scheduler



State Transition (2)

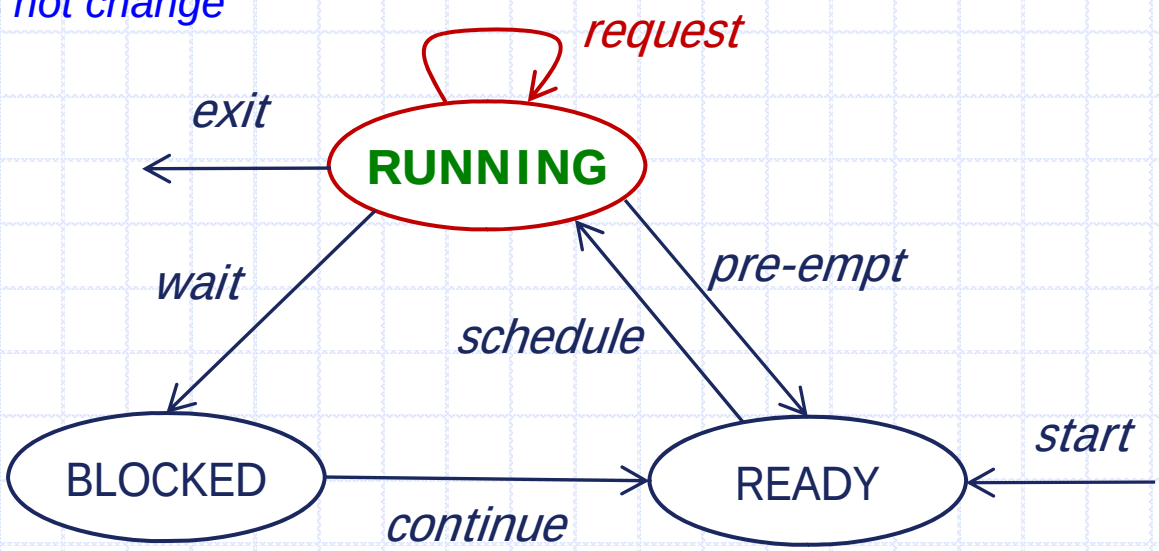
- the process remains in the READY state until a *schedule* event occurs
- this schedule event happens when the scheduler determines that it is the processes turn to use the CPU
- the schedule initiates a context switch and the CPU is given to the process
- the processes state is changed to **RUNNING**



State Transition (3)

- the process executes on the CPU, where one of three things can occur:
  1. the process could *request* an available resource
  2. the process could request an unavailable resource
  3. the process could start an operation and decide to wait
  4. the scheduler could decide that the process is out of time and pre-empt it
  5. the process could terminate

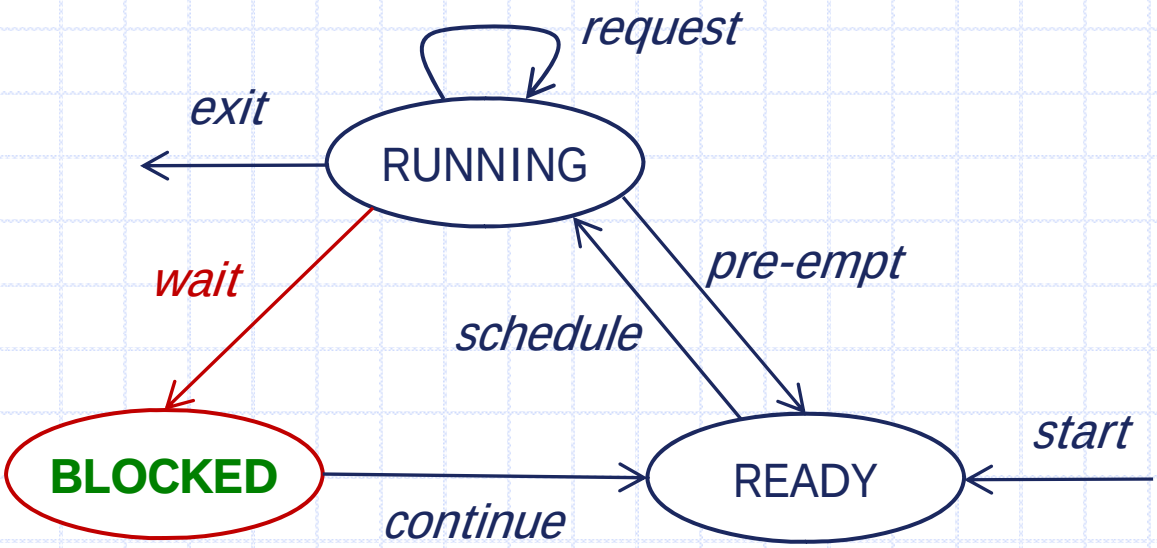
  Here we consider case 1. Since the resource is allocated immediately, the state does not change



State Transition (4)

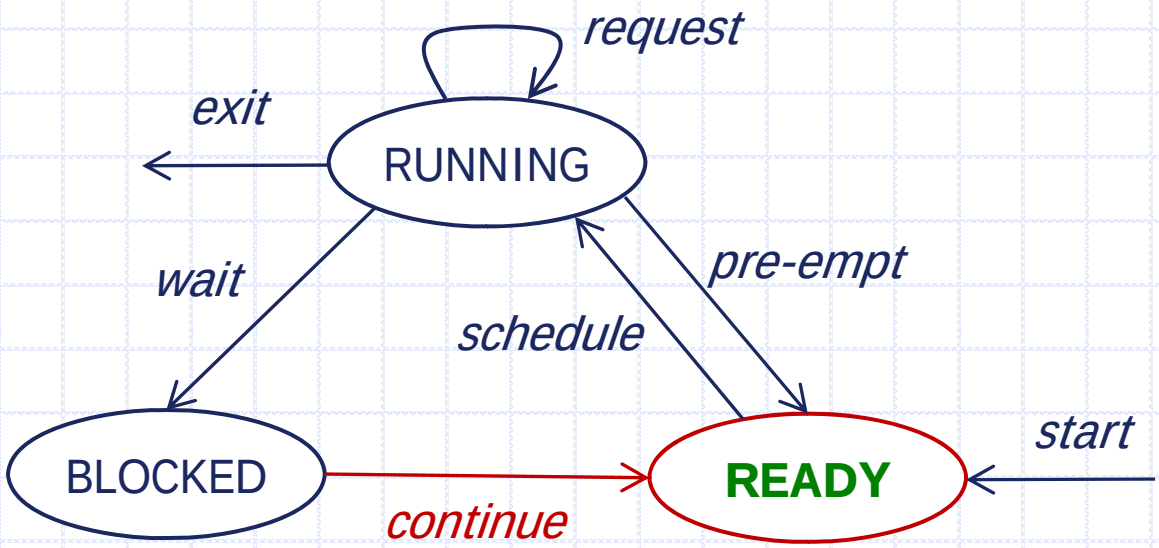
- the process is executing on the CPU, and ...
  2. the process requests a unavailable resource, or
  3. the process starts an IO operation and decides it give up the CPU an wait for the IO to complete

  *In both cases the **wait** event occurred ... the new state is **BLOCKED***



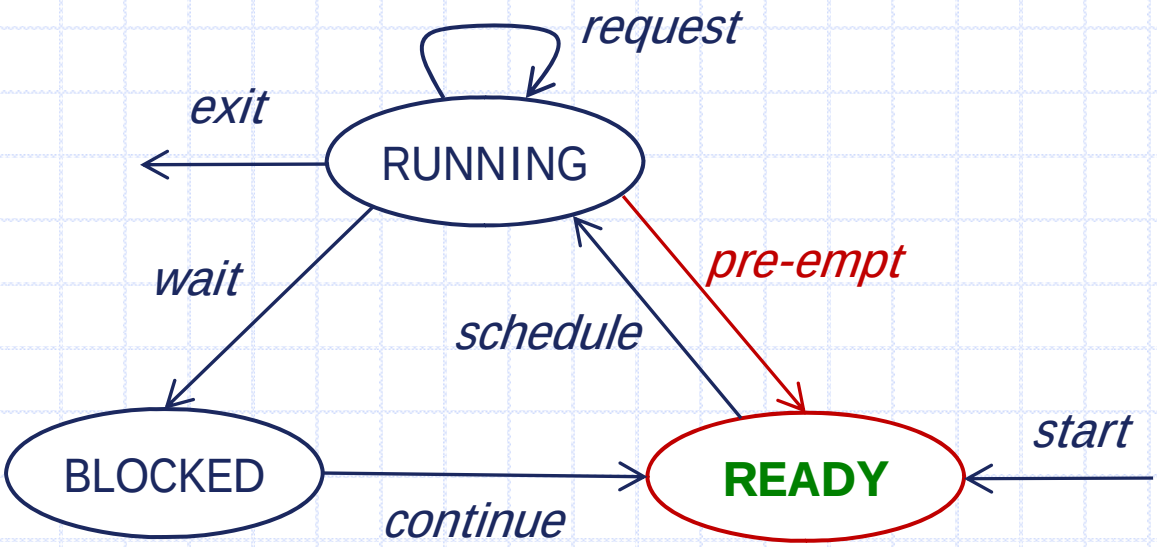
State Transition (5)

- the process remains in the BLOCKED state until the request or event it is waiting on is complete
- for example, if the process was waiting on IO it would be waiting for an interrupt from the IO device to be serviced by the ISR
- when the event occurs (ie: interrupt is received) the process can *continue*:
  - the process state is changed to **READY**
  - the process is placed in the schedulers queue again



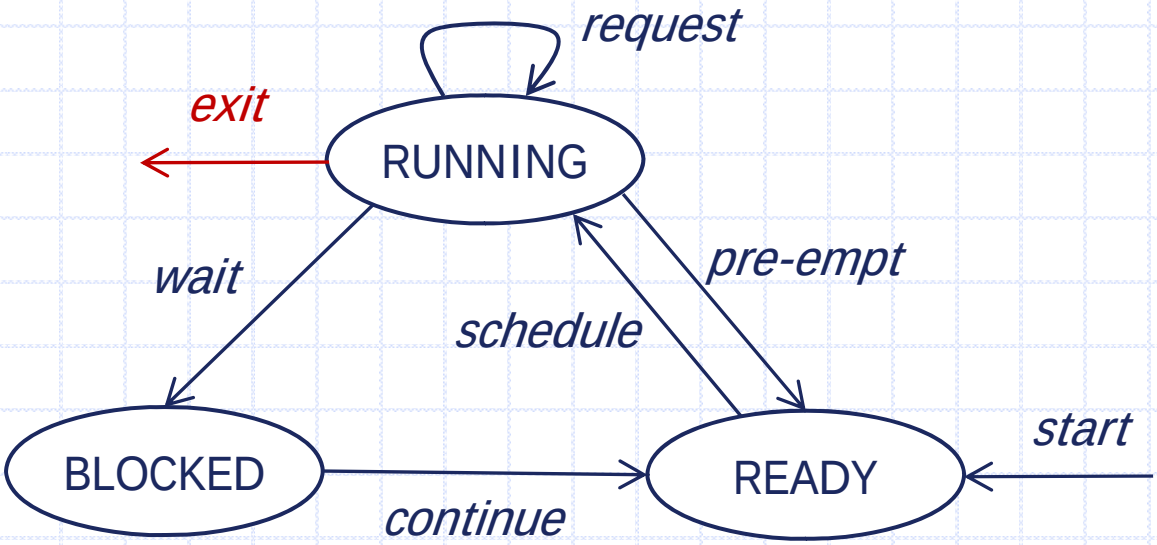
State Transition (6)

- assume a process is RUNNING, and the schedule decides to *pre-empt* it
  - the process state is changed to **READY**
  - the process is placed in the schedulers queue again
  - all processes that are in the READY state are just waiting for the scheduler to allocate the CPU to them again

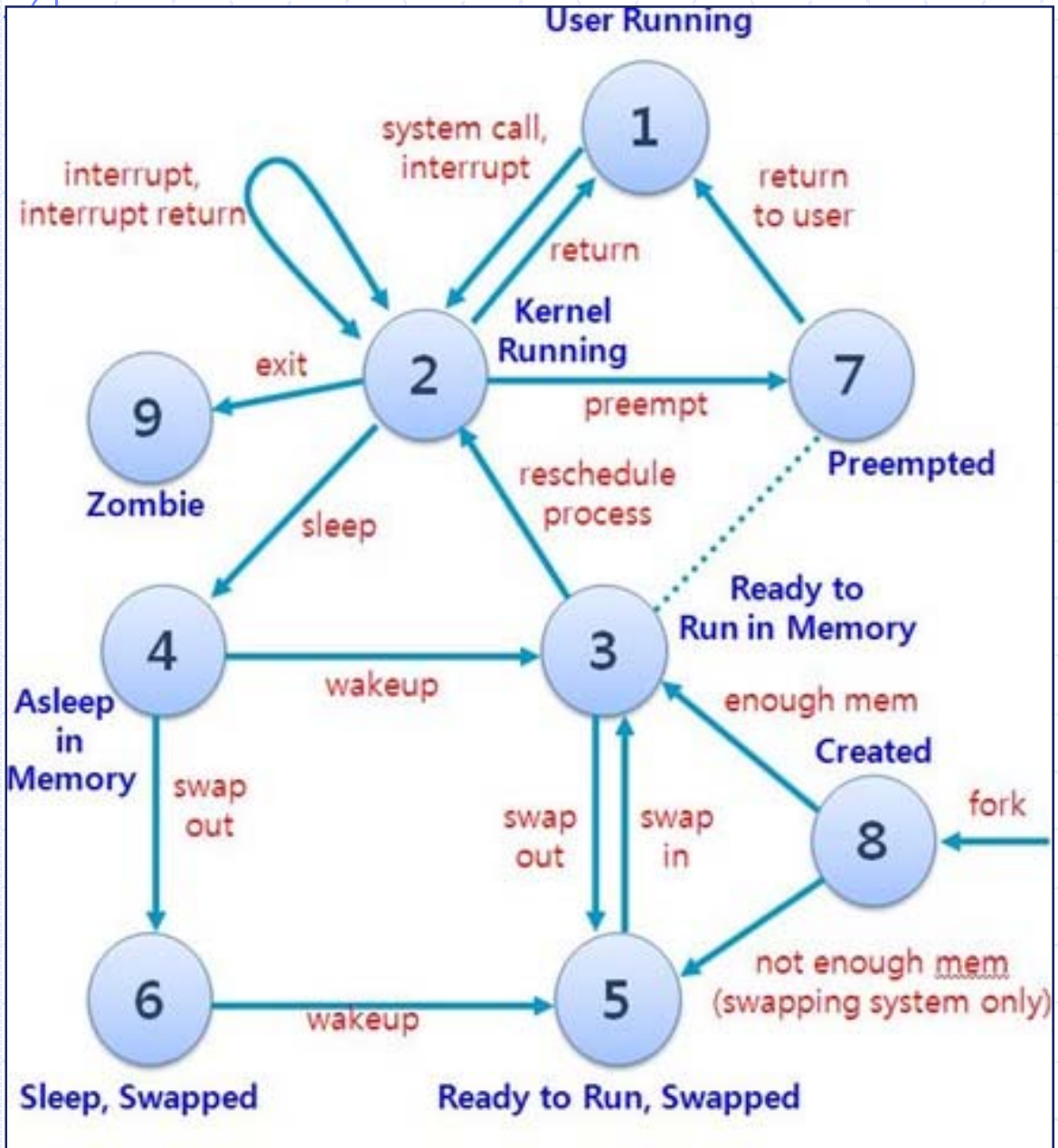


State Transition (7)

- the final case is where a process terminates
- we assume the process terminates itself (for example by issuing the *exit* command)
- the *exit* event will cause the CPU to release all of the resources and memory allocated to the process
- once the process is removed, the scheduler will go back to its queue to decide what to run next



Unix State Transitions



- in a real system, the number of states may be higher, however...
- most of these states are essentially one of:
  - running
  - ready
  - blocked
- zombie is a bit weird
  - this happens when a process has terminated, but still has an entry in the process table
  - this entry stays around until the parent reads the exit status

Process State Questions

9. A process that is in the blocked state and then has its resource request fulfilled will go to the following state:
   a) Blocked
   b) Done
   c) Ready
   d) Running
   e) Start

   – ***READY ... it gets put back in the queue to be scheduled***

Process State Questions

10. Consider the most basic state diagram (ie: the one with only 3 states). What is required for a process p_i to move from state "Ready" to state "Running"

a) p_i must cause a "Schedule" transition.
b) p_i must cause a "Done" transition.
c) p_i must cause a "request" transition.
d) p_i must cause a "Start" transition.
e) None of the above.

hint: this is sort of a trick question

- ***none of the above***

- ***it will be schedule transition, but the scheduler will cause this to happen - not p_i***

Interrupts and ISR's (revisited)

Review:

- *interrupts are generated by hardware*
  - the interrupt controller receives (marshals) interrupts from various devices, and feeds them to the CPU (sets the interrupt line HIGH)
- interrupts are processed by device specific Interrupt Service Routines (ISR's)
- the addresses of the ISR's are stored at a predetermined location in memory, known as the interrupt vector

- *to process an interrupt, the CPU must jump to the location of the ISR and start executing ....*

  ... how does this happen?

  ... what happens to the running process?

Interrupt Processing

- the initial processing of the interrupt is done without switching contexts ... here is what happens ...
  - the hardware (interrupt logic) basically performs a function call by
    - pushing the PC, PSW and a few registers onto the current stack
    - look up the address of the ISR in the interrupt vector
    - jump to the start of the ISR
  - the ISR begins by running a small assembler program (*interrupt handler*) to
    - save the current context in the PCB
      - this is just like when it performs a context switch except
      - some information is retrieved from the current stack
    - set up a new stack for the device specific ISR code to run in
    - run the device specific ISR
  - the device specific ISR continues the processing by ...
    - finding the process that is waiting for the interrupt (probably in a queue of processes waiting on the device)
    - changing the state of the waiting process from BLOCKED to READY
    - put the unblocked process in the schedulers queue
  - now the scheduler decides which process to run and performs a real context switch to start that process running

Programmatic Description of ISR

note: this is not real code; it is code that is written to describe how this might happen

```
interruptHandler() {
    disable_interrupts();
    saveProcessorState();
    for(i=0; i<NumberOfDevices; i++)
        if(device[i].done) goto deviceHandler[i];
}

// assume deviceHandler[i] contains he address
// of the following ISR (for a printer)

printerISR() {
   if (myStatus.count == 0) {          // more bytes to print?
      unblock_user();                  // move process to READY queue
   } else {
      *printerDataReg = myBuffer[i]; // send next char to printer
      myStatus.count--;                // decrement #chars remaining
      i++;                             // increment buffer pointer
   }
   acknowledge_interrupt();            // enable interrupts again
   return_to_scheduler();              // let sched decide what to run
}
```

Interrupt Handler Questions

11. Assume that a process is executing in user space when a keyboard interrupt is received for a different process. Arrange the following actions in the order the occur.

a) find the process that is waiting for the interrupt
b) look up an address in the interrupt vector
c) put the newly unblocked process in the schedulers queue
d) run the device specific ISR
e) save the PSW of the interrupted process in the process table

- ***b e d a c***

Interrupt Handler Questions

12 Which of the following statements are TRUE about interrupt processing?

a) all interrupts from all devices are disabled when handling an interrupt
   - ***yes, this is true***

b) an interrupt vector is basically just an array of addresses
   - ***yes, this is true***

c) initial interrupt processing is done without switching contexts
   - ***true - even the current processes stack is used for temp storage***

d) once the ISR is finished, control is returned to the interrupted process
   - ***no - control is passed to the scheduler***

e) none of the above are true

The very end.

- ... the very end ...