Architectural Document

Operating System Routines

Whenever user calls system calls, the operating system has its own routines to handle user requests. Every call hand over control to SVC() which uses following function to get result of system call and return to user.

1. **OsCreateProcess(char \*process\_name, void \*test, long \*p\_id, long priority )**

This function creates PCB for process and gets context from Z502MakeContext (). It saves context in PCB and adds that PCB in process table. This function also adds PCB on ready queue to run. This routine returns ERROR if bad parameters like illegal priority and duplicate process name have given. It allows to create only MAX\_NUMBER\_OF\_PROCESSES (10) processes otherwise returns ERROR. Routine also sets DEFAULT\_PRIORITY (40) if priority is not given. On successful operation it returns process id for recent created process.

1. **GetProcessID(char \*process\_name, long \*id)**

This routine gets id of process from process table if process exists and returns it with SUCCESS otherwise ERROR.

1. **TerminateProc(long id)**

When user gives -1 as ID, SVC calls this routine to remove process entry from process table as well as from queues. Process returns SUCCESS after successful termination.

(Note: -1 means only terminates itself whereas -2 means forcefully terminates whole processes though there are some processes on ready queue).

1. **ChangePriority(long id, long priority)**

This function changes priority of given process. If ID is -1, it means to change priority of itself. Otherwise it finds process on ready queue or waiting queue and changes priority. It returns ERROR for illegal priority and ID as well as ID which does not exist.

1. **SuspendProcess(long id)**

This function suspends process of given ID by adding PCB on suspended queue and returns SUCCESS. It gives ERROR for non-existent process ID or on request of suspending already suspended process. When id is -1, it suspends itself. Puts itself on suspended queue, calls GiveUpCPU() and after calls Dispatcher() to run next process which is ready to run and does not return back until it is resumed.

1. **ResumeProcess(long id)**

This function resumes process for given id by taking off PCB from suspended queue and puts it on ready queue to run. It throws ERROR for illegal id or non-existent id or when tries to resume unsuspended process.

1. **StartTimer(long delay)**

When a request comes to sleep process, this routine is called. It calls Z502DelayTimer for interrupt to occur in future. Calculates delay time and adds PCB of current running process with its delay tome on waiting queue. It calls GiveUpCPU() and then call Dispatcher() to schedule next process from ready queue. This function returns back from Dispatcher() whenever it gets time to run after time delay.

1. **SendMessage(long target\_id, long length, char \*msg)**

Sends message in MessageBox for target process. It also saves target and actual sender id of process with length of message which just have sent. Routine returns ERROR if send buffer is too small to send message or target process ID does not present or illegal message length. The LEGAL\_MESSAGE\_LENGTH is 64. Process can send message only if message length <= LEGAL\_MESSAGE\_LENGTH. This allows to send only MAX\_NUMBER\_OF\_MESSAGES (10) otherwise returns ERROR. When process is done with sending, routine checks whether target process is waiting for message or not. If it is, then it resumes process by adding PCB to ready queue from waitingtorecieve queue. If target id is -1 then resumes head of waitingtorecieve queue. This is because -1 means message is broadcast and any process can receive a message.

1. **RecieveMessage(long source\_id, long lenght\_of\_recieved\_msg, char \*msg\_buffer,long \*lenght\_of\_sent\_msg,** **long \*sender\_id)**

This function checks message is available or not. If it is not then it suspends itself by putting PCB of itself onto waitingtorecieve queue, calls GiveUpCPU() and then Dispatcher(). Process is suspended until process with source id sends message to it. If message is available then process receives it and returns message, length of message and actual sender id with SUCCESS. Deletes message from MessageBox after recieving. This function returns ERROR for illegal source id or if lenght\_of\_recieved\_msg is less than message length.

The operating system uses following function to complete a task of above functions:

1. **SetPCBContents(char \*process, long p\_id, PCB \*currentPCB, long priority)**

Sets all attributes of PCB in this function and returns process id.

1. **int sDuplicateProcess(char** \*process\_name**)**

The function checks whether other process has similar name. If found it returns 0 otherwise returns 1.

1. **CheckParentProcess(char \*name)**

This routine checks name in array which has all entries of user test. This function is used by OsCreateProcess() to do switch context at the beginning of program.

1. **CheckLegalID(long id)**

The function is used by many routines to check process is exist on process table or not. If id is found then returns SUCCESS otherwise returns ERROR.

1. **SuspendToReceive(long id)**

This routine is used by ReceiveMessage() if there is no message available for process. Function adds current running PCB onto waitingtoreceive queue, call GiveUpCPU() and after calls Dispatcher() to run next process because current running process is suspended.

1. **ResumeToReceive(long id)**

This routine is called by SendMessage() to make ready to run to process with given ID. It removes PCB from waitingtoreceive and adds on ready queue.

1. **DeleteReceivedMSG(INT32 pos)**

This routine is used by ReceiveMessage() after receiving message. It deletes message from MessageBox so other process can send message.

The operating system uses following function to support multiprocessing environment.

1. **Dispatcher()**

Dispatcher is a main function for operating system. It helps to have multiprocessing environment. Whenever any process is terminated or suspended or interrupted for any hardware service, operating system call this function. If there is nothing to run on ready queue, it waits until interrupt handler is occurred to make process ready to run. When process is available, it removes a PCB from head of ready queue and does switch context to it. It causes to run a thread which transfers control to be passed to location where PCB context points.

1. **MakeReady\_To\_Run()**

This routine is used by interrupt handler. This function makes interrupted process ready to run. Currently, there is only one interrupt in the operating system which is caused because request is made to Z502DelayTimer. This routine takes off PCB from waiting queue whose delay time is already passed and puts PCB on ready queue to run.

1. **GiveUPCPU()**

This routine give up CPU when process is suspended for any hardware service. The function calls Z502Idle if there is nothing to run on ready queue. If there is atleast one process on ready queue it just gives up CPU and goes to Dispatcher (). Basically this increases CPU utilization.

1. **SetStatePrinter\_Mode(char \*process)**

This function prints current states of operating system for specific action if isStatePrinter is true. This global variable is set at the beginning of program by OsCreateProcess() while creating process for user test program.

Queue related function which operating system needs to store PCB during various system calls.

1. **AddInWaitingQueue(PCB \*current\_PCB)**

This function stores PCB according to delay time. PCB which has less delay time, is head of queue.

1. **AddInReadyQueue(PCB \*current\_PCB)**

This function saves PCB according to priority. PCB which has highest priority is saved as a first element or finds a position on queue and sits there. If priority is same it saves PCB as criteria.

Following 3 functions saves PCB according to FCFS criteria.

1. **AddInReadyQueueByPriority(PCB \*current\_PCB)**
2. **AddInSuspenedQueue(PCB \*current\_PCB)**
3. **AddInWaitingReceiveQueue(PCB \*current\_PCB)**

The following function 6 - 11 removes PCB from queue and returns it.

1. **PCB \*DequeueHeadFromReadyQueue()**
2. **PCB \*DequeueHeadFromWaitingQueue()**
3. **PCB\* DequeueFromReadyQueue(long pid)**
4. **PCB\* DequeueFromWaitingQueue(long pid)**
5. **PCB\* DequeueFromSuspenedQueue(long pid)**
6. **PCB\* DequeueFromWatingReceiveQueue(long pid)**
7. **int IsSuspended(long pid)**

This function returns 1 if process is already present on suspended queue otherwise returns 0.

1. **PrintQueue (ReadyQueue \*head);**

This function prints element of queue pointed by given head.

1. **int IsEmptyQueue(struct Queue \*head);**

Returns 0 if queue is not empty otherwise returns 1.

Following function gets elements form queue to set variables of state\_printer to print current states of processes.

1. **GetReadyQElements();**
2. **GetWaitingQElements();**
3. **GetWaitingToReceiveQElements();**
4. **SetQElementsToDefault();**

**Function**

1. **Interrupt\_handler()**

If process is waiting for any hardware service, that time this function is invoked to handle interrupt. When SLEEP system call requests to idle for delay time, this function is invoked. Interrupt\_handler realizes what causes it. For SLEEP system call, it is TIMER\_INTERRUPT and therefore it calls MakeReady\_To\_Run() routine to wake up process and make it ready to run.

1. **Fault\_handler()**

In test1k user try to use PRIVILEGED\_INSTRUCTION from user mode, so hardware invokes this routine. It displays error to user and exits whole program. You can see details of this function in bace.c .

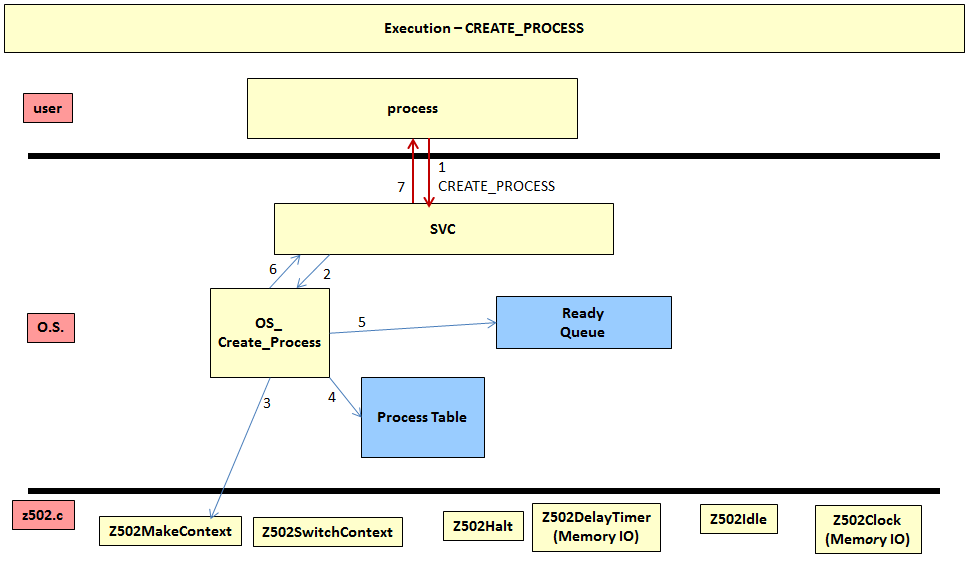
***Execution of System Calls***

Every system calls causes control to be passed to operating system’s SVC() which then calls above operating system routine and completes tasks of system call and returns SUCCESS or ERROR.

1. **Execution of CREATE\_PROCESS :**

When process from user mode calls CREATE\_PROCESS system call and it transfer control to SVC() routine.

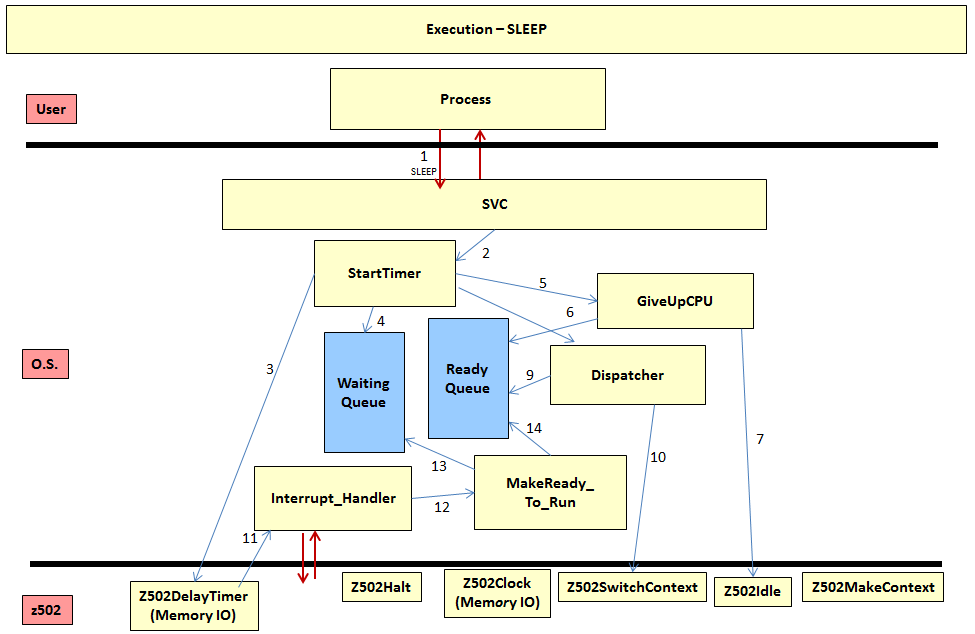
SVC() then calls OsCreateProcess() which does error checking and returns immediately with ERROR if any found. If each user given parameter is ok to create process then it calls Z502MakeContext() to get context and store new process PCB in PCB table. OsCreateProcess() enqueues PCB of newly created process in Ready Queue according to priority. OsCreateProcess() returns success to SVC() if it creating process operation is successful and returns. (NOTE: It is also responsible to do context switch if operating system call requests to create process eg. Test1b or etc).

****

**Figure 1: Execution of CREATE\_SYSTEM call**

1. **Execution of SLEEP system call:**

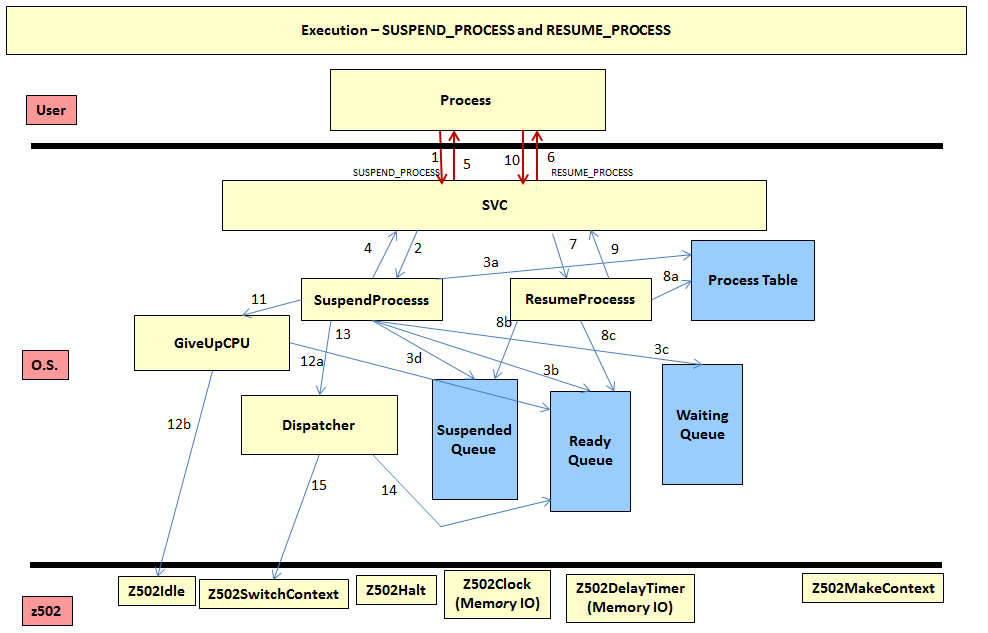
When process from user mode calls SLEEP system call, it gives control to SVC (). SVC () calls StartTimer () routine which calls Z502DelayYTimer to give the request for a future interrupt. It calculates time delay and adds current running process PCB with time delay into waiting queue and calls GiveUpCPU (). GiveUpCPU () gives up CPU for current process and calls Z509Idle() if there is nothing to run on ready queue and remains idle until interrupt handler occurs, otherwise returns back to StartTimer() immediately. StartTimer() calls Dispatcher() to schedule a another process from ready queue. Dispatcher() try to get a lock on ready queue and takes off a PCB from ready queue and does switch context on dequeued PCB which causes new thread to run. This new thread passes control to location where PCB context points. In between execution of new process, Interrupt handler thread gets chance to run. Interrupt handler checks what causes it to run. Interrupt handler calls MakeReady\_To\_Run () which checks time delay of all PCB. It takes off all PCB from waiting queue whose delay is passed and put those PCB onto ready queue to run. The process which called SLEEP returns back to location from where is it interrupted as soon as it gets chance to run.



**Figure 2: Execution of SLEEP call**

1. **Execution of SUSPEND\_PROCESS and RECEIVE\_PROCESS system call:**

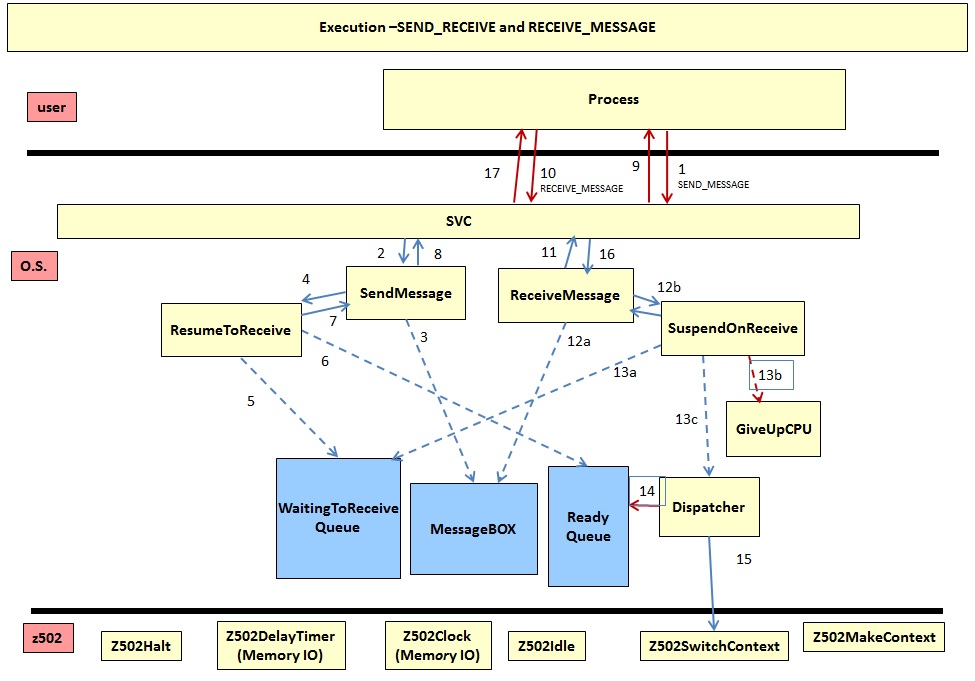
Important note here is when process suspends itself it puts itself on suspened queue, calls GiveUpCPU() and after to Dispacther (). This process remains suspened until it is resumed by anyone. (Please see the figure 3 for flow)



**Figure 3: Execution of SUSPEND\_PROCESS and RECEIVE\_PROCESS call**

1. **Execution of SEND\_MESSAGE and RECEIVE\_MESSAGE system call:**

The sending process uses non- blocking send. Therefore it needs a MessageBox to send message. Process saves message on MessageBox with its own id and target process id so only target process can receive it. When receiving process checks MessageBox, it matches its id with target and source id with message’s source id. If it equals then and then only it can receive message otherwise it suspends itself until message is sent to it. If message is broadcast, any process can read message and deletes after receiving. The flow of SEND\_MESSAGE and RECEIVE\_MESSAGE is as follows.



**Figure 4: Execution of SEND\_MESSAGE and RECEIVE\_MESSAGE call**

1. **Execution of TERMINATE\_PROCESS system call:**

When process calls TERMINATE\_PROCESS system call with -2, SVC() calls Z502Halt() and terminates program forcefully. It does not check any other process is alive or not. Id = -1 means, terminate itself. Therefore, calls TerminateProc() to remove PCB.

***Execution of User’s all test routine:***

You can see flow of test1b, test1c, test1f, test1j in images folder.