5103 Project 1

Generated by Doxygen 1.7.6.1

Tue Feb 21 2012 15:18:54

Contents

Chapter 1

Class Index

1.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 4

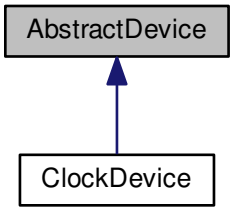
Class Documentation

4.1 AbstractDevice Class Reference

An abstract device class.

`#include <abstract_device.h>`

Inheritance diagram for AbstractDevice:



Public Member Functions

- virtual void setTimer (int time)=0

  Set the length of the timer for this device.

- virtual void disarm ()=0

  Disarm the timer set from a previous call to setTimer.

4.1.1 Detailed Description

This was initially gonig to ge the abstract class for all devices such as the clock, character and block devices but it turned out this was not the desired interface for everything. This is why the cAbsQueuedDevice class was made. This class remains here because it is used by the clock device however it is no longer necessary.

4.1.2 Member Function Documentation

4.1.2.1 void AbstractDevice::setTimer (int *time*) [pure virtual]

Parameters

int	time It is up to the implementation what the scale for this time is. It will likely be milliseconds or more.

Implemented in ClockDevice.

The documentation for this class was generated from the following file:

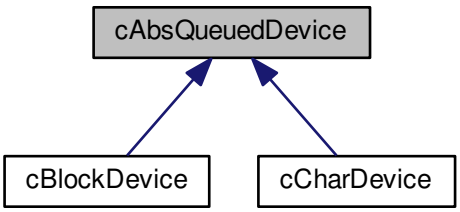
- include/devices/abstract_device.h

4.2 cAbsQueuedDevice Class Reference

Abstract class for a device which queues requests.

`#include <queued_device.h>`

Inheritance diagram for cAbsQueuedDevice:



Public Member Functions

- virtual void setDefaultTime (int usec)=0

Set the default timer length for this device.

- virtual void scheduleDevice (ProcessInfo *)=0

  Schedule a process for a device interrupt.

- virtual ProcessInfo * timerFinished ()=0

  Called when a device timer has gone off.

- virtual int queueLength ()=0

  Returns the number of devices waiting on the device.

4.2.1 Member Function Documentation

4.2.1.1 void cAbsQueuedDevice::scheduleDevice (ProcessInfo *) [pure virtual]

If there are no other waiting processes then schedule the interrupt. If there is a pending interrupt then queue the process.

Parameters

*ProcessInfo**	Process that wants to block on the device. This must be saved and returned to the kernel when its interrupt has been received.

Warning

This must be synchronized with timerFinished because signal handlers will cause these methods to be called asynchronously.

Implemented in cCharDevice, and cBlockDevice.

4.2.1.2 void cAbsQueuedDevice::setDefaultTime (int *usec*) [pure virtual]

Subsequent calls to schedule the device for a process should use this default time. - This is convenience function. A class defined default can be defined statically or in the constructor.

Parameters

int	usec default microsecond length for device timer.

Implemented in cCharDevice, and cBlockDevice.

4.2.1.3 ProcessInfo * cAbsQueuedDevice::timerFinished ( ) [pure virtual]

The signal handler in the kernel will call this method when a device's timer has completed. The method should then return the waiting process and then schedule an interrupt for the next device queued, if any.

Returns

ProcessInfo* Process that was waiting for the device I/O to complete. It will now be unblocked by the kernel/scheduler.

Warning

Must be synchronized with scheduleDevice. Failing to do so could leave a process blocked indefinitely.

Implemented in cCharDevice, and cBlockDevice.

The documentation for this class was generated from the following file:

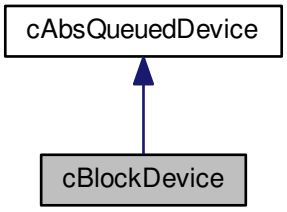
- include/devices/queued_device.h

4.3 cBlockDevice Class Reference

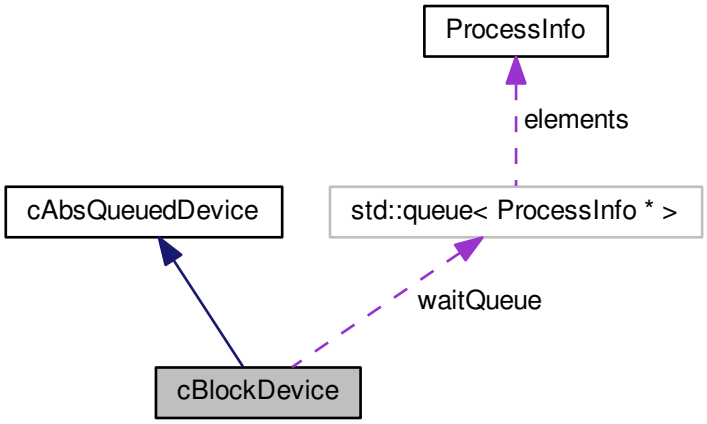
Queued block device.

`#include <block_device.h>`

Inheritance diagram for cBlockDevice:



Collaboration diagram for cBlockDevice:



Public Member Functions

- **cBlockDevice** (int usec)
- void setDefaultTime (int usec)

  Set the default timer length for this device.
- void scheduleDevice (ProcessInfo *)

  Schedule a process for a device interrupt.
- ProcessInfo * timerFinished ()

  Called when a device timer has gone off.
- int queueLength ()

  Returns the number of devices waiting on the device.

Private Attributes

- queue< ProcessInfo * > **waitQueue**
- pthread_mutex_t **deviceLock**

4.3.1 Member Function Documentation

4.3.1.1 void cBlockDevice::scheduleDevice (ProcessInfo *) [virtual]

If there are no other waiting processes then schedule the interrupt. If there is a pending interrupt then queue the process.

Parameters

ProcessInfo∗	Process that wants to block on the device. This must be saved and returned to the kernel when its interrupt has been received.

Warning

This must be synchronized with timerFinished because signal handlers will cause these methods to be called asynchronously.

Implements cAbsQueuedDevice.

Referenced by cKernel::boot().

4.3.1.2 void cBlockDevice::setDefaultTime (int *usec*) [virtual]

Subsequent calls to schedule the device for a process should use this default time. - This is convenience function. A class defined default can be defined statically or in the constructor.

Parameters

int	usec default microsecond length for device timer.

Implements cAbsQueuedDevice.

4.3.1.3 ProcessInfo ∗ cBlockDevice::timerFinished ( ) [virtual]

The signal handler in the kernel will call this method when a device's timer has completed. The method should then return the waiting process and then schedule an interrupt for the next device queued, if any.

Returns

ProcessInfo∗ Process that was waiting for the device I/O to complete. It will now be unblocked by the kernel/scheduler.

Warning

Must be synchronized with scheduleDevice. Failing to do so could leave a process blocked indefinitely.

Implements cAbsQueuedDevice.

The documentation for this class was generated from the following files:

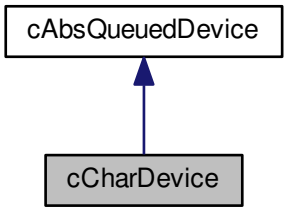
- include/devices/block_device.h
- src/devices/block_device.cpp

4.4 cCharDevice Class Reference

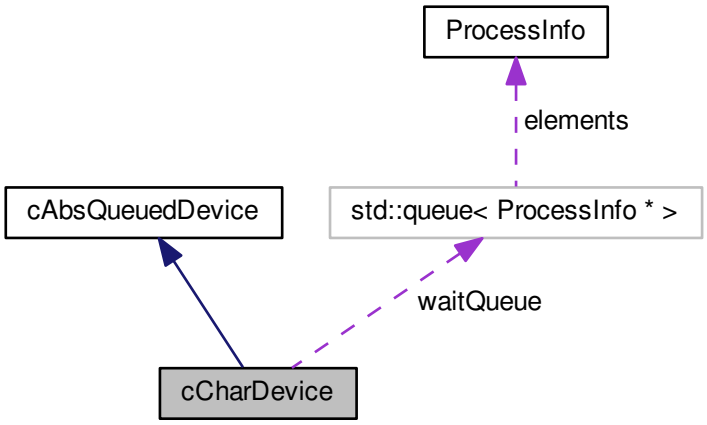
Queued char device.

`#include <char_device.h>`

Inheritance diagram for cCharDevice:



Collaboration diagram for cCharDevice:



Public Member Functions

- **cCharDevice** (int usec)

- void setDefaultTime (int usec)

  Set the default timer length for this device.
- void scheduleDevice (ProcessInfo *)

  Schedule a process for a device interrupt.
- ProcessInfo * timerFinished ()

  Called when a device timer has gone off.
- int queueLength ()

  Returns the number of devices waiting on the device.

Private Attributes

- queue< ProcessInfo * > **waitQueue**
- pthread_mutex_t **deviceLock**

4.4.1 Member Function Documentation

4.4.1.1 void cCharDevice::scheduleDevice (ProcessInfo *) [virtual]

If there are no other waiting processes then schedule the interrupt. If there is a pending interrupt then queue the process.

Parameters

*ProcessInfo**	Process that wants to block on the device. This must be saved and returned to the kernel when its interrupt has been received.

Warning

This must be synchronized with timerFinished because signal handlers will cause these methods to be called asynchronously.

Implements cAbsQueuedDevice.

Referenced by cKernel::boot().

4.4.1.2 void cCharDevice::setDefaultTime (int *usec*) [virtual]

Subsequent calls to schedule the device for a process should use this default time. - This is convenience function. A class defined default can be defined statically or in the constructor.

Parameters

int	usec default microsecond length for device timer.

Implements cAbsQueuedDevice.

4.4.1.3 ProcessInfo * cCharDevice::timerFinished ( ) [virtual]

The signal handler in the kernel will call this method when a device's timer has completed. The method should then return the waiting process and then schedule an interrupt for the next device queued, if any.

Returns

ProcessInfo* Process that was waiting for the device I/O to complete. It will now be unblocked by the kernel/scheduler.

Warning

Must be synchronized with scheduleDevice. Failing to do so could leave a process blocked indefinitely.

Implements cAbsQueuedDevice.

The documentation for this class was generated from the following files:

- include/devices/char_device.h
- src/devices/char_device.cpp

4.5 cCPU Class Reference

A class for emulating a simple cpu.

`#include <cpu.h>`

Public Member Functions

- void **initTraceLog** ()
- void **initClockPulse** (pthread_mutex_t *_pulseLock, pthread_cond_t *_pulseCond)
- void setText (char *text)

  Set the program text.
- void setMaxPC (unsigned int newMax)

  Set the max PC value.
- void setUserMode ()

  Set the cpu back into user mode.
- unsigned int getSetPC (unsigned int newPC)

  Get/Set the program counter.
- int getSetVC (int newVC)

  Get/Set the VC.
- uint16_t getSetPSW (uint16_t newPSW)

  Get/Set the PSW.

- uint16_t getPSW ()

  Get the Program Status Word.
- void setPSW (uint16_t newPSW)

  Set a new value for the PSW.
- char * getParam (int num)

  Get execution parameters from the cpu.
- char getOpcode ()

  Get the current Opcode.
- void run ()

  Start execution.
- void executePrivSet (int num, int &clockTick)

  Execute set number of privleged instructions.

Public Attributes

- pidType **pid**

Private Member Functions

- int **tokenizeLine** ()

Private Attributes

- int **clockTick**
- bool KMode

  Kernel Mode bit.
- unsigned int **PC**
- unsigned int **maxPC**
- int **VC**
- uint16_t PSW

  Program Status Word.
- char * execText

  Text data for currently executing process.
- char tokenBuffer [2][MAX_PARAM_SIZE]

  Holds the tokenized execution parameters.
- char Opcode

  Holds the current Opcode.
- FILE * **traceStream**
- pthread_mutex_t * pulseLock

  Used when executing privileged set to synchronize with clock.
- pthread_cond_t * pulseCond

  Used in conjunction with pulseLock.

4.5.1 Detailed Description

This class emulates the internals of a very simple cpu with two main registers, PC and VC. In addition, it has other state for handling system calls and program exceptions.

4.5.2 Member Function Documentation

4.5.2.1 void cCPU::executePrivSet (int *num,* int & *clockTick*)

When the kernel receives a syscall for device I/O it will call this function to "simulate" executing kernel mode ops.

Parameters

int	num Number of privleged instructios to execute.
int&	clockTick Reference to the clockTick counter in the kenel.

Referenced by cKernel::boot().

4.5.2.2 char cCPU::getOpcode ( )

Get the current Opcode in the cpu. This is used by the kernel to determine which system call as being made. Used in conjunciton with cCPU::getParam the kernel can process system calls.

Referenced by cKernel::boot().

4.5.2.3 char * cCPU::getParam (int *num*)

Fetch the given execution paramter from the cpu's internal buffer. When an instruction is encountered that has parameters associated with it, the cpu tokenizes them and places it in an internal buffer. This function is mainly used by the kernel in handling system calls.

Parameters

num	Must be less than MAX_PARAMS (currently 2)

Returns

Returns a char* which points to a string of at most MAX_PARAM_SIZE - 1 bytes.

Referenced by cKernel::boot().

4.5.2.4 uint16_t cCPU::getPSW ( )

Returns the program status word which is a unsigned 16-bit integer type with flags from ePSW set. These are used by the kernel to make action decisions.

Referenced by cKernel::boot().

4.5.2.5 unsigned int cCPU::getSetPC (unsigned int *newPC*)

Get the current value for the program counter and then set its value to the given parameter. This is useful for swapping out process values.

Referenced by cKernel::swapProcesses().

4.5.2.6 uint16_t cCPU::getSetPSW (uint16_t *newPSW*)

Get the current value for the PSW and set its value to the given parameter.

Referenced by cKernel::swapProcesses().

4.5.2.7 int cCPU::getSetVC (int *newVC*)

Get the current value for VC and set its value to the given paramter. This is useful for swapping out process values.

Referenced by cKernel::boot(), and cKernel::swapProcesses().

4.5.2.8 void cCPU::run ( )

Once all appropriate process data is entered by the kernel this function is called to start execution. Any time control needs to be returned to the kernel this function will return with the appropriate PSW flags set for the kernel to act on.

Referenced by cKernel::boot().

4.5.2.9 void cCPU::setMaxPC (unsigned int *newMax*)

This sets the maxPC 'register' in the cpu. This is used when parsing commands to ensure that the cpu does not fall of the end of the process' text segment.

Referenced by cKernel::swapProcesses().

4.5.2.10 void cCPU::setPSW (uint16_t *newPSW*)

Used by the kernel to reset the PSW after a system call. Any process execution which returns to the kernel but does not terminate the process should reset the PSW so subsequent exceptions/terminations are not lost by stray PSW values.

Referenced by cKernel::boot().

4.5.2.11 void cCPU::setText (char * *text*)

Point the cpu to the text data for the running process. This text is indexed using the program counter (PC).

Parameters

text	Program text pointer. assert(text != NULL)

Referenced by cKernel::swapProcesses().

4.5.2.12 void cCPU::setUserMode ( )

This is used by the kernel after the kernel has finished servicing a process' kernel mode request (syscall).

Referenced by cKernel::swapProcesses().

4.5.3 Member Data Documentation

4.5.3.1 bool cCPU::KMode [private]

Set upon system calls.

Referenced by executePrivSet(), run(), and setUserMode().

The documentation for this class was generated from the following files:

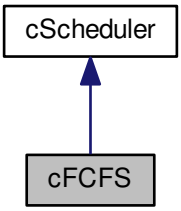
- include/cpu.h
- src/cpu.cpp

4.6 cFCFS Class Reference

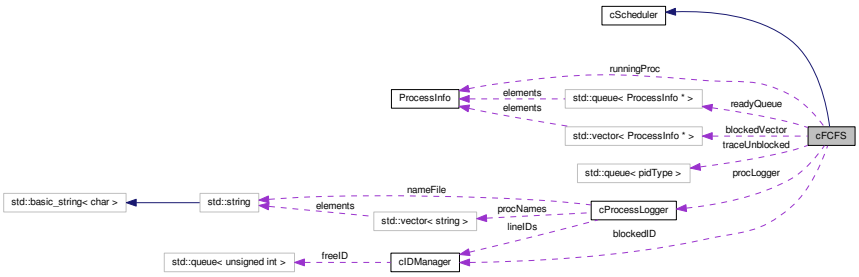
First-Come-First-Serve Scheduler.

`#include <fcfs.h>`

Inheritance diagram for cFCFS:



Collaboration diagram for cFCFS:



Public Member Functions

- void initProcScheduleInfo (ProcessInfo ∗)

  Initialize scheduler specific information within the ProcessInfo struct.
- void addProcess (ProcessInfo ∗)

  Transfer control of process state and scheduling.
- void setBlocked (ProcessInfo ∗)

  Set a process into a blocked state.
- void unblockProcess (ProcessInfo ∗)

  Unblock a process and make it ready.
- void removeProcess (ProcessInfo ∗)

  Remove a process from the control of the scheduler.
- ProcessInfo ∗ getNextToRun ()

  Query the scheduler for next process to run.
- pidType numProcesses ()

How many processes are in the scheduler.

- void addLogger (FILE *_logStream)

  Add an instance of the trace logger to the scheduler.
- void addProcLogger (cProcessLogger *_procLogger)

  Add instance of process logger to the scheduler.
- void printUnblocked ()

  Prints at once all processes that may have become unblocked asynchronously.

Private Attributes

- queue< ProcessInfo * > **readyQueue**
- vector< ProcessInfo * > **blockedVector**
- queue< pidType > **traceUnblocked**
- int **totalBlocked**
- ProcessInfo * **runningProc**
- pthread_mutex_t **blockedLock**
- pthread_cond_t **allBlocked**
- cIDManager **blockedID**
- FILE * **logStream**
- cProcessLogger * **procLogger**

4.6.1 Member Function Documentation

4.6.1.1 void cFCFS::addLogger (FILE *) [virtual]

The kernel passes a file stream pointer for the scheduler to print trace info such as blocking/unblocking events.

Parameters

*FILE**	File stream for trace log

See also

initLog(const char* filename)
getStream()

Implements cScheduler.

4.6.1.2 void cFCFS::addProcess (ProcessInfo *) [virtual]

After this is called, the kernel core no longer keeps track of the given process. Once the process is created and deemed ready by the kernel it is handed off here. The scheduler is then in charge of state transitions when the kernel gives it appropriate notifications.

Implementation Requirements:

- Store the process in some location. It should be recognized as ready given its location but the datastructures and organization are implementation specific.

Parameters

ProcessInfo∗	Process to add under scheduler's control

Implements cScheduler.

4.6.1.3 void cFCFS::addProcLogger (cProcessLogger ∗) [virtual]

The kernel passes this pointer so the scheduler can update process info upon process addition, state change and termination. This updates the information presented to the top process.

Parameters

cProcessLogger∗	Process logger class

See also

cProcessLogger

Implements cScheduler.

4.6.1.4 ProcessInfo ∗ cFCFS::getNextToRun ( ) [virtual]

After this function is called, it should be assumed by any scheduler implementation that the kernel will run the given process (unless otherwise notified). The currently running process should implicitly be considered for running next (again).

If there are processes left but all are blocked. This function should block until it receives a signal that a process is unblocked.

Implementation Requirements:

- Call printUnblocked() to update the trace log after a clock step.
- Block if there are $>$ 0 processes but none can run.
- A process listed as running within the scheduler should be treated as ready when this method is called.
- A process returned as the 'next to run' should be marked as being in a running state before returning.
- If there are no more remaining processes this should return NULL.

Returns

ProcessInfo* Ready process to run next. May be the same as the currenlty running one.

Warning

Must be thread safe with block and unblock methods.

Implements cScheduler.

4.6.1.5 void cFCFS::initProcScheduleInfo (ProcessInfo *) [virtual]

This method is called after the kernel has initialized all process data but before it is marked ready. This gives the scheduler an oportunity to initialize any scheduler specific data and assign it to the ProcessInfo::scheduleData member.

Implementation Requirements:

- No required actions.

Implements cScheduler.

4.6.1.6 pidType cFCFS::numProcesses ( ) [virtual]

This returns how many processes, both running and blocked, are being handled by the scheduler.

Implementation Requirements:

- Return how many processes, running and blocked, are in the scheduler.

Returns

pidType

Implements cScheduler.

4.6.1.7 void cFCFS::printUnblocked ( ) [virtual]

This method is needed to avoid mixed output in the trace logger.

Implements cScheduler.

Referenced by getNextToRun().

4.6.1.8 void cFCFS::removeProcess (ProcessInfo *) [virtual]

When a process terminates, either through normal means or an exception, the kernel will call this function to release a process from the scheduler's control. The scheduler should clean up any internal state for the process. Deallocation of process resources is left to the kernel.

Implementation Requirements:

- Scheduler should deallocate any resources it assigned to the process within the ProcessInfo::scheduleData member.
- The Scheduler should remove any pointers to the give process to avoid dereferencing a dead pointer.
- Implementations must not deallocate any memory except that mentioned above. This is handled by the kernel.
- Implementations should mark the process as terminated.

Parameters

*ProcessInfo**	Process to remove from scheduler

Implements cScheduler.

4.6.1.9 void cFCFS::setBlocked (ProcessInfo *) [virtual]

The kernel will call the scheduler with this function when the process has done an operation which causes it to block (I).

Implementation Requirements:

- Process must be marked blocked and scheduler state should be changed accordingly.
- After this call a process should not be considered for a scheduling decision

Warning

Must be thread safe. Signal handler/s may block during schedule decision.

Implements cScheduler.

4.6.1.10 void cFCFS::unblockProcess (ProcessInfo *) [virtual]

When a process has completed a blocking call the kernel will notify the scheduler that it should be unblocked. This operation should be very fast since it will likely be called from a signal handler.

Implementation Requirements:

- The process must be unblocked and marked ready. It must be available for scheduling with the next call to ::getNextToRun

Parameters

*ProcessInfo**	Process to unblock

Warning

Must be thread safe. Signal handler/s may unblock during schedule decision.

Implements cScheduler.

The documentation for this class was generated from the following files:

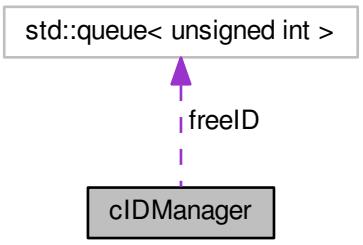
- include/scheduler/fcfs.h
- src/scheduler/fcfs.cpp

4.7 cIDManager Class Reference

A class for managing unique IDs.

`#include <id.h>`

Collaboration diagram for cIDManager:



Public Member Functions

- cIDManager (unsigned int startID=0)

Creates a new ID Manager object.

- unsigned int getID ()

  Reserves a unique ID.

- unsigned int getLowID ()

  Get a low ID.

- void returnID (unsigned int id)

  Returns an ID to the manager.

- unsigned int nextLowID ()

  See what the next low ID would be.

- unsigned int reservedIDs ()

  How many IDs have been given out.

Private Attributes

- queue< unsigned int > freeID

  Queue of returned IDs.

- unsigned int baseID

  To prevent the IDs from dropping below when being returned.

- unsigned int currentID

  Next ID to be given out.

- bool consumeQueue

  This signals that IDs have reached their max and freeIDs should be used.

4.7.1 Constructor & Destructor Documentation

4.7.1.1 cIDManager::cIDManager (unsigned int *startID* = 0)

Default start ID is 0.

4.7.2 Member Function Documentation

4.7.2.1 unsigned int cIDManager::getID ( )

Unique is in the sense that no one else is currently using it but it may have been used previously. are distributed in increasing order until UINT_MAX is reached. After this is reached, IDs are given from the queue of returned IDs. If this queue is empty then an exception is thrown.

Referenced by getLowID(), cKernel::initProcess(), cFCFS::setBlocked(), cRoundRobin-::setBlocked(), cLottery::setBlocked(), and cMultiLevel::setBlocked().

4.7.2.2 unsigned int cIDManager::getLowID ( )

When generating process PID's we use the regular getID so that process IDs continue to grow. This choice was mainly to prevent confusion when process 1 terminated and the next one to start had pid = 1. For functions which use vectors and need an ID system, it is more efficient to maintain a smaller window to keep the array small. This method provides this by preferring to return IDs from the freeID queue. Therefore, if there is an ID availabe in freeID then the total range of IDs will not grow after this function call.

Referenced by cLottery::addProcess(), and cLottery::unblockProcess().

4.7.2.3 unsigned int cIDManager::nextLowID ( )

There is not longer any purpose for this functino but I left it here for the potential functionality. The intention was to improve performance in the process logger to determine if the next ID would be right after the previous low ID. That way, if we had variable length records we wouldn't have to search from the beginning.

4.7.2.4 unsigned int cIDManager::reservedIDs ( )

Returns the number of IDs which have been reserved

4.7.2.5 void cIDManager::returnID (unsigned int *id*)

If the ID is not equal to the one last given then it is added to a 'free queue'. If it is equal to the last one reserved then the ID counter is simply decremented If this last case happens, it causes cIDManager::getID to stop consuming from the queue and return this newly availabe ID.

Referenced by cKernel::cleanupProcess(), cLottery::removeProcess(), cLottery::setBlocked(), cFCFS::unblockProcess(), cRoundRobin::unblockProcess(), cLottery::unblockProcess(), and cMultiLevel::unblockProcess().

The documentation for this class was generated from the following files:

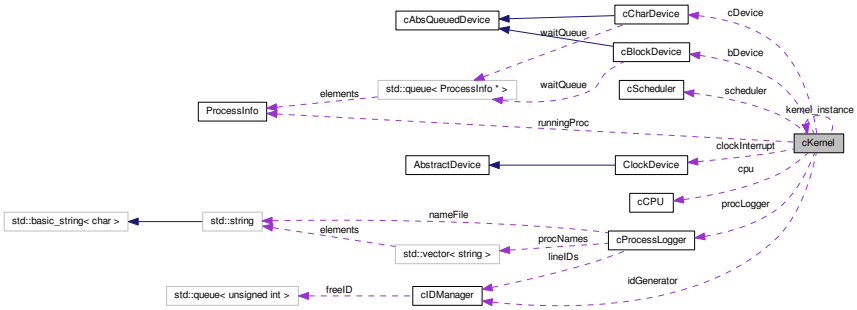
- include/utility/id.h
- src/utility/id.cpp

4.8 cKernel Class Reference

Core managing class for this simulated OS.

```
#include <kernel.h>
```

Collaboration diagram for cKernel:



Public Member Functions

- cKernel (cScheduler &)

  Default cKernel constructor.
- void boot ()

  Start the 'OS' Kernel.
- void initProcess (const char *filename, pidType parent, int priority=DEFAULT_PRIORITY)

  Initialize a Process.
- void cleanupProcess (pidType pid)

  Cleans up a terminated process.

Private Member Functions

- void swapProcesses (ProcessInfo *proc, bool switchMode=true)

  Swap a process on the cpu.
- void cleanupProcess (ProcessInfo *)

  Cleanup process memory and state.
- void sigHandler (int signum, siginfo_t *info)

  Handler for all signals.

Static Private Member Functions

- static void sig_catch (int signum, siginfo_t *info, void *context)

  Static function just for capturing signals.

Private Attributes

- FILE * **traceStream**
- cProcessLogger **procLogger**
- cCPU **cpu**
- int **clockTick**
- pthread_mutex_t intLock

  Lock for condition variable.
- pthread_cond_t intCond

  For synchronization with clocktick.
- cBlockDevice **bDevice**
- cCharDevice **cDevice**
- ClockDevice **clockInterrupt**
- pthread_t **deviceThread**
- sem_t DevSigSem

  A device interrupt has been received.
- sem_t BSigSem

  A block device interrupt has been received.
- sem_t CSigSem

  A char device interrupt has been received.
- ProcessInfo * runningProc

  Process currently on the cpu.
- cIDManager idGenerator

  For generating new process PID's.
- cScheduler & **scheduler**

Static Private Attributes

- static cKernel * **kernel_instance**

Friends

- void * deviceHandle (void *)

  Thread function for handline device interrupts.

4.8.1 Constructor & Destructor Documentation

4.8.1.1 cKernel::cKernel (cScheduler & *s*)

The default constructor initializes all internal datastructures and loads the initial program (default: 'main.trace') but does not run it.

4.8.2 Member Function Documentation

4.8.2.1 void cKernel::boot ( )

Starts the main kernel loop. The initial program is loaded and execution follows from there.

Exceptions

kernelError	

4.8.2.2 void cKernel::cleanupProcess (ProcessInfo * *proc*) [private]

This is called when a process is being removed from the system and its memory and any remaining state information needs to be cleaned up.

Referenced by boot().

4.8.2.3 void cKernel::cleanupProcess (pidType *pid*)

Cleans up any memory and kernel entries associated with the terminated process. Also removes the process from the scheduler.

4.8.2.4 void cKernel::initProcess (const char * *filename*, pidType *parent*, int *priority* = DEFAULT_PRIORITY)

Initializes a process by loading program file contents, setting default process values and adding it in a ready state to the scheduler.

Referenced by boot(), and cKernel().

4.8.2.5 void cKernel::sigHandler (int *signum*, siginfo_t * *info*) [private]

For clock interrupt signals, the handler signals on a condition variable which the main thread will be waiting on.

For block and character devices, the appropriate semaphores are incremented which will unblock the waiting thread to act on them.

This function is called by cKernel::sig_catch

See also

deviceHandle(void*)
cKernel::intCond
DevSigSem
CSigSem
BSigSem

Referenced by sig_catch().

4.8.2.6 void cKernel::swapProcesses (ProcessInfo * *proc,* bool *switchMode* = `true`) [private]

Takes the process in its parameter and swaps it with the one currently running in the cpu.

Referenced by boot().

4.8.3 Friends And Related Function Documentation

4.8.3.1 void* deviceHandle (void * *args*) [friend]

Since limited work can be done within the signal handlers and the main thread may block, a separate thread must be present to handle device interrupts.

See also

DevSigSem
CSigSem
BSigSem

Parameters

*void**	This is a pointer to the kernel instance.

Referenced by boot().

The documentation for this class was generated from the following files:

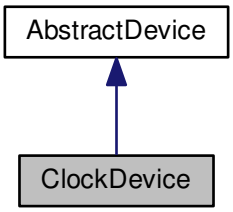
- include/kernel.h
- src/kernel.cpp

4.9 ClockDevice Class Reference

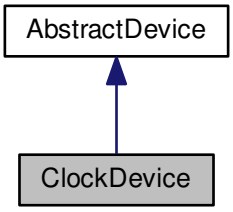
Device for generating repeated clock interrupts.

`#include <clock_device.h>`

Inheritance diagram for ClockDevice:



Collaboration diagram for ClockDevice:



Public Member Functions

- void setTimer (int usec)

  Set the timer to go off.
- void disarm ()

  Disarm the timer.
- int getTime ()

  Get how much time is remaining.

4.9.1 Member Function Documentation

4.9.1.1 void ClockDevice::disarm ( ) [virtual]

Tries to disarm the timer.

Exceptions

std::string	error message

Implements AbstractDevice.

4.9.1.2 int ClockDevice::getTime ( )

Returns the remaining time until a signal is produced.

Returns

int Time left in microseconds

4.9.1.3 void ClockDevice::setTimer (int *usec*) [virtual]

Sets timer to send signal CLOCKSIG in usec microseconds.

Parameters

usec	Time in microseconds.

Exceptions

std::string	error message

Implements AbstractDevice.

Referenced by cKernel::boot().

The documentation for this class was generated from the following files:

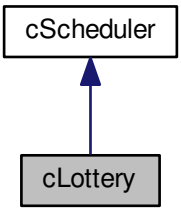
- include/devices/clock_device.h
- src/devices/clock_device.cpp

4.10 cLottery Class Reference

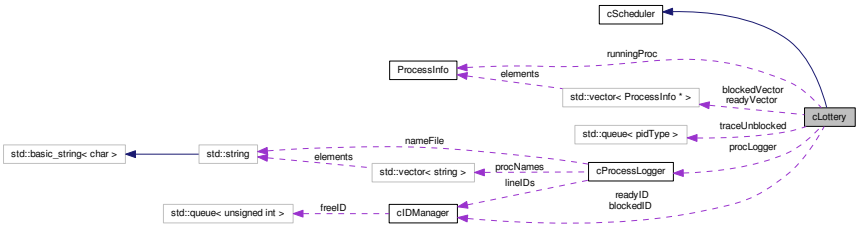
Lottery Scheduler.

`#include <lottery.h>`

Inheritance diagram for cLottery:



Collaboration diagram for cLottery:



Public Member Functions

- void initProcScheduleInfo (ProcessInfo *)

  Initialize scheduler specific information within the ProcessInfo struct.
- void addProcess (ProcessInfo *)

  Transfer control of process state and scheduling.
- void setBlocked (ProcessInfo *)

  Set a process into a blocked state.
- void unblockProcess (ProcessInfo *)

  Unblock a process and make it ready.
- void removeProcess (ProcessInfo *)

  Remove a process from the control of the scheduler.
- ProcessInfo * getNextToRun ()

  Query the scheduler for next process to run.
- pidType numProcesses ()

  How many processes are in the scheduler.

- void addLogger (FILE *_logStream)

  Assign the file pointer to the logStream data field.
- void addProcLogger (cProcessLogger *_procLogger)

  Assign the cProcessLogger pointer to the procLogger data field.
- void printUnblocked ()

  Removes each PID from the traceUnblocked queue and prints it to the trace logger.

Private Attributes

- vector< ProcessInfo * > **readyVector**
- vector< ProcessInfo * > **blockedVector**
- queue< pidType > **traceUnblocked**
- int **totalReady**
- int **totalBlocked**
- int **totalTickets**
- ProcessInfo * **runningProc**
- pthread_mutex_t **blockedLock**
- pthread_cond_t **allBlocked**
- cIDManager **blockedID**
- cIDManager **readyID**
- FILE * **logStream**
- cProcessLogger * **procLogger**

4.10.1 Member Function Documentation

4.10.1.1 void cLottery::addProcess (ProcessInfo *) [virtual]

After this is called, the kernel core no longer keeps track of the given process. Once the process is created and deemed ready by the kernel it is handed off here. The scheduler is then in charge of state transitions when the kernel gives it appropriate notifications.

Implementation Requirements:

- Store the process in some location. It should be recognized as ready given its location but the datastructures and organization are implementation specific.

Parameters

*ProcessInfo**	Process to add under scheduler's control

Implements cScheduler.

4.10.1.2 ProcessInfo * cLottery::getNextToRun ( ) [virtual]

After this function is called, it should be assumed by any scheduler implementation that the kernel will run the given process (unless otherwise notified). The currently running process should implicitly be considered for running next (again).

If there are processes left but all are blocked. This function should block until it receives a signal that a process is unblocked.

Implementation Requirements:

- Call printUnblocked() to update the trace log after a clock step.
- Block if there are $>$ 0 processes but none can run.
- A process listed as running within the scheduler should be treated as ready when this method is called.
- A process returned as the 'next to run' should be marked as being in a running state before returning.
- If there are no more remaining processes this should return NULL.

Returns

ProcessInfo* Ready process to run next. May be the same as the currenlty running one.

Warning

Must be thread safe with block and unblock methods.

Implements cScheduler.

4.10.1.3 void cLottery::initProcScheduleInfo (ProcessInfo *) [virtual]

This method is called after the kernel has initialized all process data but before it is marked ready. This gives the scheduler an oportunity to initialize any scheduler specific data and assign it to the ProcessInfo::scheduleData member.

Implementation Requirements:

- No required actions.

Implements cScheduler.

4.10.1.4 pidType cLottery::numProcesses ( ) [virtual]

This returns how many processes, both running and blocked, are being handled by the scheduler.

Implementation Requirements:

- Return how many processes, running and blocked, are in the scheduler.

Returns

pidType

Implements cScheduler.

4.10.1.5 void cLottery::removeProcess (ProcessInfo *) [virtual]

When a process terminates, either through normal means or an exception, the kernel will call this function to release a process from the scheduler's control. The scheduler should clean up any internal state for the process. Deallocation of process resources is left to the kernel.

Implementation Requirements:

- Scheduler should deallocate any resources it assigned to the process within the ProcessInfo::scheduleData member.
- The Scheduler should remove any pointers to the give process to avoid dereferncing a dead pointer.
- Implementations must not deallocate any memory except that mentioned above. This is handled by the kernel.
- Implementations should mark the process as terminated.

Parameters

*ProcessInfo**	Process to remove from scheduler

Implements cScheduler.

4.10.1.6 void cLottery::setBlocked (ProcessInfo *) [virtual]

The kernel will call the scheduler with this function when the process has done an operation which causes it to block (I).

Implementation Requirements:

- Process must be marked blocked and scheduler state should be changed accordingly.
- After this call a process should not be considered for a scheduling decision

Warning

Must be thread safe. Signal handler/s may block during schedule decision.

Implements cScheduler.

4.10.1.7 void cLottery::unblockProcess (ProcessInfo *) [virtual]

When a process has completed a blocking call the kernel will notify the scheduler that it should be unblocked. This operation should be very fast since it will likely be called from a signal handler.

Implementation Requirements:

- The process must be unblocked and marked ready. It must be available for scheduling with the next call to ::getNextToRun

Parameters

*ProcessInfo**	Process to unblock

Warning

Must be thread safe. Signal handler/s may unblock during schedule decision.

Implements cScheduler.

The documentation for this class was generated from the following files:

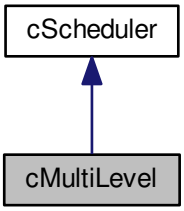
- include/scheduler/lottery.h
- src/scheduler/lottery.cpp

4.11 cMultiLevel Class Reference

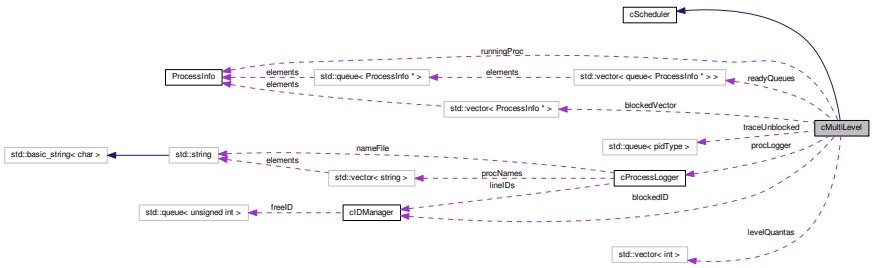
Multi-Level Queue Scheduler.

`#include <multi_level.h>`

Inheritance diagram for cMultiLevel:



Collaboration diagram for cMultiLevel:



Public Member Functions

- void initProcScheduleInfo (ProcessInfo ∗)

  Initialize scheduler specific information within the ProcessInfo struct.
- void addProcess (ProcessInfo ∗)

  Transfer control of process state and scheduling.
- void setBlocked (ProcessInfo ∗)

  Set a process into a blocked state.
- void unblockProcess (ProcessInfo ∗)

  Unblock a process and make it ready.
- void removeProcess (ProcessInfo ∗)

  Remove a process from the control of the scheduler.
- ProcessInfo ∗ getNextToRun ()

  Query the scheduler for next process to run.
- pidType numProcesses ()

How many processes are in the scheduler.

- void addLogger (FILE *_logStream)

  Add an instance of the trace logger to the scheduler.

- void addProcLogger (cProcessLogger *_procLogger)

  Add instance of process logger to the scheduler.

- void printUnblocked ()

  Prints at once all processes that may have become unblocked asynchronously.

- void **printLevels** ()

Private Attributes

- int **currentLevel**
- int **quantaUsed**
- int **totalReady**
- int **totalBlocked**
- vector< queue< ProcessInfo * > > **readyQueues**
- vector< int > **levelQuantas**
- vector< ProcessInfo * > **blockedVector**
- queue< pidType > **traceUnblocked**
- ProcessInfo * **runningProc**
- pthread_mutex_t **blockedLock**
- pthread_cond_t **allBlocked**
- cIDManager **blockedID**
- FILE * **logStream**
- cProcessLogger * **procLogger**

4.11.1 Member Function Documentation

4.11.1.1 void cMultiLevel::addLogger (FILE *) [virtual]

The kernel passes a file stream pointer for the scheduler to print trace info such as blocking/unblocking events.

Parameters

*FILE**	File stream for trace log

See also

initLog(const char* filename)
getStream()

Implements cScheduler.

4.11.1.2 void cMultiLevel::addProcess (ProcessInfo *) [virtual]

After this is called, the kernel core no longer keeps track of the given process. Once the process is created and deemed ready by the kernel it is handed off here. The scheduler is then in charge of state transitions when the kernel gives it appropriate notifications.

Implementation Requirements:

- Store the process in some location. It should be recognized as ready given its location but the datastructures and organization are implementation specific.

Parameters

*ProcessInfo**	Process to add under scheduler's control

Implements cScheduler.

4.11.1.3 void cMultiLevel::addProcLogger (cProcessLogger *) [virtual]

The kernel passes this pointer so the scheduler can update process info upon process addition, state change and termination. This updates the information presented to the top process.

Parameters

*cProcessLogger**	Process logger class

See also

cProcessLogger

Implements cScheduler.

4.11.1.4 ProcessInfo * cMultiLevel::getNextToRun ( ) [virtual]

After this function is called, it should be assumed by any scheduler implementation that the kernel will run the given process (unless otherwise notified). The currently running process should implicitly be considered for running next (again).

If there are processes left but all are blocked. This function should block until it receives a signal that a process is unblocked.

Implementation Requirements:

- Call printUnblocked() to update the trace log after a clock step.
- Block if there are $>$ 0 processes but none can run.

- A process listed as running within the scheduler should be treated as ready when this method is called.
- A process returned as the 'next to run' should be marked as being in a running state before returning.
- If there are no more remaining processes this should return NULL.

Returns

ProcessInfo* Ready process to run next. May be the same as the currenlty running one.

Warning

Must be thread safe with block and unblock methods.

Implements cScheduler.

4.11.1.5 void cMultiLevel::initProcScheduleInfo (ProcessInfo *) [virtual]

This method is called after the kernel has initialized all process data but before it is marked ready. This gives the scheduler an oportunity to initialize any scheduler specific data and assign it to the ProcessInfo::scheduleData member.

Implementation Requirements:

- No required actions.

Implements cScheduler.

4.11.1.6 pidType cMultiLevel::numProcesses ( ) [virtual]

This returns how many processes, both running and blocked, are being handled by the scheduler.

Implementation Requirements:

- Return how many processes, running and blocked, are in the scheduler.

Returns

pidType

Implements cScheduler.

4.11.1.7 void cMultiLevel::printUnblocked ( ) [virtual]

This method is needed to avoid mixed output in the trace logger.

Implements cScheduler.

Referenced by getNextToRun().

4.11.1.8 void cMultiLevel::removeProcess (ProcessInfo *) [virtual]

When a process terminates, either through normal means or an exception, the kernel will call this function to release a process from the scheduler's control. The scheduler should clean up any internal state for the process. Deallocation of process resources is left to the kernel.

Implementation Requirements:

- Scheduler should deallocate any resources it assigned to the process within the ProcessInfo::scheduleData member.
- The Scheduler should remove any pointers to the give process to avoid dereferncing a dead pointer.
- Implementations must not deallocate any memory except that mentioned above. This is handled by the kernel.
- Implementations should mark the process as terminated.

Parameters

*ProcessInfo**	Process to remove from scheduler

Implements cScheduler.

4.11.1.9 void cMultiLevel::setBlocked (ProcessInfo *) [virtual]

The kernel will call the scheduler with this function when the process has done an operation which causes it to block (I).

Implementation Requirements:

- Process must be marked blocked and scheduler state should be changed accordingly.
- After this call a process should not be considered for a scheduling decision

Warning

Must be thread safe. Signal handler/s may block during schedule decision.

Implements cScheduler.

4.11.1.10 void cMultiLevel::unblockProcess (ProcessInfo *) [virtual]

When a process has completed a blocking call the kernel will notify the scheduler that it should be unblocked. This operation should be very fast since it will likely be called from a signal handler.

Implementation Requirements:

- The process must be unblocked and marked ready. It must be available for scheduling with the next call to ::getNextToRun

Parameters

*ProcessInfo**	Process to unblock

Warning

Must be thread safe. Signal handler/s may unblock during schedule decision.

Implements cScheduler.

The documentation for this class was generated from the following files:

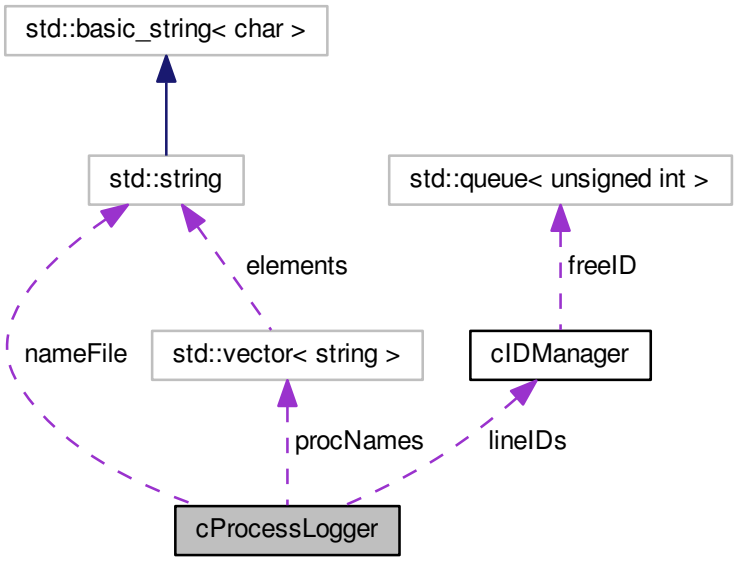
- include/scheduler/multi_level.h
- src/scheduler/multi_level.cpp

4.12 cProcessLogger Class Reference

Class specifically for logging process state information.

`#include <process_logger.h>`

Collaboration diagram for cProcessLogger:



Public Member Functions

- **cProcessLogger** (const char *file)
- void **addProcess** (ProcessInfo *, const char *)
- void **rmProcess** (ProcessInfo *)
- void **writeProcessInfo** (ProcessInfo *)

Private Member Functions

- void **addToVector** (FILE *)

Private Attributes

- cIDManager **lineIDs**
- string **nameFile**
- int **procLogFD**
- FILE * **procLogStream**
- int **lineSize**
- char **outputBuffer** [MAX_LINE_LENGTH]

- char **emptyBuffer** [MAX_LINE_LENGTH]
- int **previousID**
- int **listenSock**
- pthread_t **nameReqListener**
- vector< string > **procNames**
- pthread_mutex_t **logWriteLock**

Friends

- void * **nameSockFn** (void *)

4.12.1 Detailed Description

In order for monitoring programs such as top to function the kernel and its associated modules must export process information. This class logs information for each process to a file named by its pid. This is inspired by the unix /proc filesystem (although memory mapped files aren't being used). This allows the kernel to easily update only those processes which have changed.

The documentation for this class was generated from the following files:

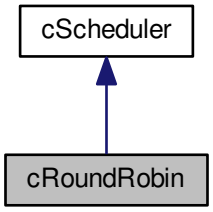
- include/utility/process_logger.h
- src/utility/process_logger.cpp

4.13 cRoundRobin Class Reference

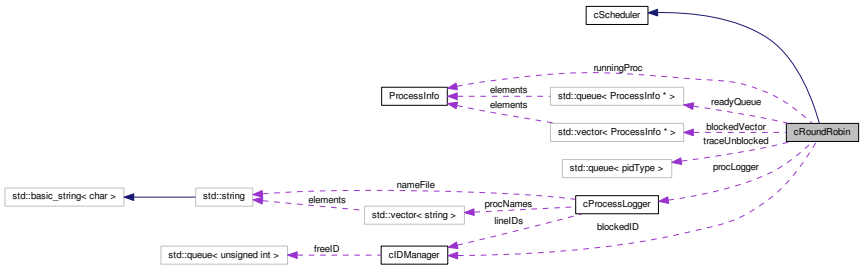
Round Robin Scheduler.

`#include <round_robin.h>`

Inheritance diagram for cRoundRobin:



Collaboration diagram for cRoundRobin:



Public Member Functions

- void initProcScheduleInfo (ProcessInfo ∗)

  Initializes a new roundRobinInfo struct.
- void addProcess (ProcessInfo ∗)

  Adds a process to the back of the readyQueue.
- void setBlocked (ProcessInfo ∗)

  Adds a process to the blocked vector using an id created with the ID Manager.
- void unblockProcess (ProcessInfo ∗)

  Sets the process to be ready again and adds it to the back of the queue.
- void removeProcess (ProcessInfo ∗)

  Sets the process state to be terminated and resets clockTicksUsed to 0.
- ProcessInfo ∗ getNextToRun ()

  If a proces is currently executing and clockTicksUsed is greater than the QUANTUM, stop the current process from executing futher and add it to the back of the ready queue.
- pidType numProcesses ()

  Count up the number of processes in the readyQueue, proceses that are blocked, and add 1 if a process is currently running.
- void addLogger (FILE ∗_logStream)

  Assign the file pointer to the logStream data field.
- void addProcLogger (cProcessLogger ∗_procLogger)

  Assign the cProcessLogger pointer to the procLogger data field.
- void printUnblocked ()

  Removes each PID from the traceUnblocked queue and prints it to the trace logger.

Private Attributes

- queue< ProcessInfo ∗ > **readyQueue**
- vector< ProcessInfo ∗ > **blockedVector**

- queue< pidType > **traceUnblocked**
- int **totalBlocked**
- int **clockTicksUsed**
- ProcessInfo * **runningProc**
- pthread_mutex_t **blockedLock**
- pthread_cond_t **allBlocked**
- cIDManager **blockedID**
- FILE * **logStream**
- cProcessLogger * **procLogger**

4.13.1 Member Function Documentation

4.13.1.1 ProcessInfo * cRoundRobin::getNextToRun ( ) [virtual]

If clockTicksUsed is less than the QUANTUM, increment clockTicksUsed and return the currently executing process to the kernel to keep running. If a new process needs to be scheduled, grab the top of the ready queue and increment clockTicksUsed.

Implements cScheduler.

4.13.1.2 void cRoundRobin::removeProcess (ProcessInfo * *proc*) [virtual]

Frees the roundRobinInfo struct

Implements cScheduler.

4.13.1.3 void cRoundRobin::setBlocked (ProcessInfo * *proc*) [virtual]

Sets runningProc to NULL and the state of the process to blocked. Sets clockTicksUsed to 0 as a new process will be picked to run. Increment the count of blocked processes.

Implements cScheduler.

4.13.1.4 void cRoundRobin::unblockProcess (ProcessInfo * *proc*) [virtual]

Returns the blocked index id back to the ID Manager for future reuse. Decrement the count of blocked processes. Add the blocked process id to the traceUnblocked queue.

Implements cScheduler.

The documentation for this class was generated from the following files:

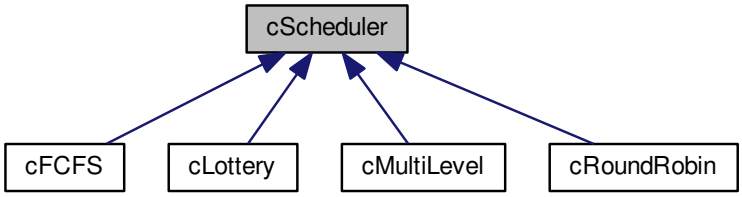
- include/scheduler/round_robin.h
- src/scheduler/round_robin.cpp

4.14 cScheduler Class Reference

Abstract Interface for Schedulers.

`#include <scheduler.h>`

Inheritance diagram for cScheduler:



Public Member Functions

- virtual void initProcScheduleInfo (ProcessInfo *)=0

  Initialize scheduler specific information within the ProcessInfo struct.
- virtual void addProcess (ProcessInfo *)=0

  Transfer control of process state and scheduling.
- virtual void setBlocked (ProcessInfo *)=0

  Set a process into a blocked state.
- virtual void unblockProcess (ProcessInfo *)=0

  Unblock a process and make it ready.
- virtual void removeProcess (ProcessInfo *)=0

  Remove a process from the control of the scheduler.
- virtual ProcessInfo * getNextToRun ()=0

  Query the scheduler for next process to run.
- virtual pidType numProcesses ()=0

  How many processes are in the scheduler.
- virtual void addLogger (FILE *)=0

  Add an instance of the trace logger to the scheduler.
- virtual void addProcLogger (cProcessLogger *)=0

  Add instance of process logger to the scheduler.
- virtual void printUnblocked ()=0

  Prints at once all processes that may have become unblocked asynchronously.

4.14.1 Member Function Documentation

4.14.1.1 void cScheduler::addLogger (FILE *) [pure virtual]

The kernel passes a file stream pointer for the scheduler to print trace info such as blocking/unblocking events.

Parameters

*FILE**	File stream for trace log

See also

initLog(const char* filename)
getStream()

Implemented in cMultiLevel, cLottery, cRoundRobin, and cFCFS.

Referenced by cKernel::cKernel().

4.14.1.2 void cScheduler::addProcess (ProcessInfo *) [pure virtual]

After this is called, the kernel core no longer keeps track of the given process. Once the process is created and deemed ready by the kernel it is handed off here. The scheduler is then in charge of state transitions when the kernel gives it appropriate notifications.

Implementation Requirements:

- Store the process in some location. It should be recognized as ready given its location but the datastructures and organization are implementation specific.

Parameters

*ProcessInfo**	Process to add under scheduler's control

Implemented in cMultiLevel, cLottery, cRoundRobin, and cFCFS.

Referenced by cKernel::initProcess().

4.14.1.3 void cScheduler::addProcLogger (cProcessLogger *) [pure virtual]

The kernel passes this pointer so the scheduler can update process info upon process addition, state change and termination. This updates the information presented to the top process.

Parameters

cProcess-Logger∗	Process logger class

See also

cProcessLogger

Implemented in cMultiLevel, cLottery, cRoundRobin, and cFCFS.

Referenced by cKernel::cKernel().

4.14.1.4 ProcessInfo ∗ cScheduler::getNextToRun ( ) [pure virtual]

After this function is called, it should be assumed by any scheduler implementation that the kernel will run the given process (unless otherwise notified). The currently running process should implicitly be considered for running next (again).

If there are processes left but all are blocked. This function should block until it receives a signal that a process is unblocked.

Implementation Requirements:

- Call printUnblocked() to update the trace log after a clock step.
- Block if there are $>$ 0 processes but none can run.
- A process listed as running within the scheduler should be treated as ready when this method is called.
- A process returned as the 'next to run' should be marked as being in a running state before returning.
- If there are no more remaining processes this should return NULL.

Returns

ProcessInfo∗ Ready process to run next. May be the same as the currenlty running one.

Warning

Must be thread safe with block and unblock methods.

Implemented in cMultiLevel, cLottery, cRoundRobin, and cFCFS.

Referenced by cKernel::boot().

4.14.1.5 void cScheduler::initProcScheduleInfo (ProcessInfo *) [pure virtual]

This method is called after the kernel has initialized all process data but before it is marked ready. This gives the scheduler an oportunity to initialize any scheduler specific data and assign it to the ProcessInfo::scheduleData member.

Implementation Requirements:

- No required actions.

Implemented in cMultiLevel, cLottery, cRoundRobin, and cFCFS.

Referenced by cKernel::initProcess().

4.14.1.6 pidType cScheduler::numProcesses ( ) [pure virtual]

This returns how many processes, both running and blocked, are being handled by the scheduler.

Implementation Requirements:

- Return how many processes, running and blocked, are in the scheduler.

Returns

pidType

Implemented in cMultiLevel, cLottery, cRoundRobin, and cFCFS.

Referenced by cKernel::boot(), and cKernel::cKernel().

4.14.1.7 void cScheduler::printUnblocked ( ) [pure virtual]

This method is needed to avoid mixed output in the trace logger.

Implemented in cMultiLevel, cLottery, cRoundRobin, and cFCFS.

4.14.1.8 void cScheduler::removeProcess (ProcessInfo *) [pure virtual]

When a process terminates, either through normal means or an exception, the kernel will call this function to release a process from the scheduler's control. The scheduler should clean up any internal state for the process. Deallocation of process resources is left to the kernel.

Implementation Requirements:

- Scheduler should deallocate any resources it assigned to the process within the ProcessInfo::scheduleData member.

- The Scheduler should remove any pointers to the give process to avoid derefernncing a dead pointer.
- Implementations must not deallocate any memory except that mentioned above. This is handled by the kernel.
- Implementations should mark the process as terminated.

Parameters

ProcessInfo∗	Process to remove from scheduler

Implemented in cMultiLevel, cLottery, cRoundRobin, and cFCFS.

Referenced by cKernel::cleanupProcess().

4.14.1.9 void cScheduler::setBlocked (ProcessInfo ∗) [pure virtual]

The kernel will call the scheduler with this function when the process has done an operation which causes it to block (I).

Implementation Requirements:

- Process must be marked blocked and scheduler state should be changed accordingly.
- After this call a process should not be considered for a scheduling decision

Warning

Must be thread safe. Signal handler/s may block during schedule decision.

Implemented in cMultiLevel, cLottery, cRoundRobin, and cFCFS.

Referenced by cKernel::boot().

4.14.1.10 void cScheduler::unblockProcess (ProcessInfo ∗) [pure virtual]

When a process has completed a blocking call the kernel will notify the scheduler that it should be unblocked. This operation should be very fast since it will likely be called from a signal handler.

Implementation Requirements:

- The process must be unblocked and marked ready. It must be available for scheduling with the next call to ::getNextToRun

Parameters

*ProcessInfo**	Process to unblock

Warning

Must be thread safe. Signal handler/s may unblock during schedule decision.

Implemented in cMultiLevel, cLottery, cRoundRobin, and cFCFS.

The documentation for this class was generated from the following file:

- include/scheduler/scheduler.h

4.15 fcfsInfo Struct Reference

Struct containing process info specific for FCFS scheduling.

`#include <fcfs.h>`

Public Attributes

- unsigned int blockedIndex

  Index position in blocked vector.

The documentation for this struct was generated from the following file:

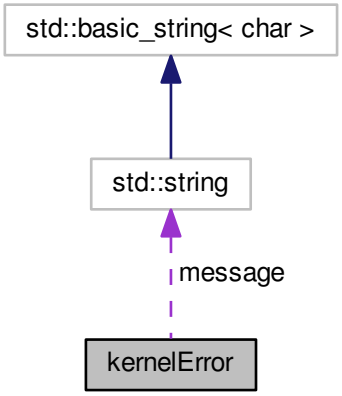
- include/scheduler/fcfs.h

4.16 kernelError Struct Reference

Struct containing kernel crash information.

`#include <kernel.h>`

Collaboration diagram for kernelError:



Public Attributes

- string **message**

4.16.1 Detailed Description

When the kernel crashes, important information is placed in here and then handled by the main function in init.cpp.

The documentation for this struct was generated from the following file:

- include/kernel.h

4.17 lotteryInfo Struct Reference

Struct containing process info specific for Lottery scheduling.

`#include <lottery.h>`

Public Attributes

- unsigned int readyIndex

  Index position in ready vector.

- unsigned int blockedIndex

  Index position in blocked vector.

The documentation for this struct was generated from the following file:

- include/scheduler/lottery.h

4.18 ProcessInfo Struct Reference

Structure for containing process state and data.

`#include <process.h>`

Public Attributes

- pidType **procFileLine**
- unsigned int **parent**
- unsigned int **pid**
- unsigned int **startCPU**
- unsigned int **totalCPU**
- eProcState **state**
- uint16_t **PSW**
- int **priority**
- unsigned int **PC**
- int **VC**
- char * **processText**
- void * scheduleData

  Scheduler specific data.
- unsigned long **memory**

4.18.1 Detailed Description

This struture is created in the kernel when a process is initialized. It contains all process data needed for execution and for the kernel/scheduler to make desciions on it.

4.18.2 Member Data Documentation

4.18.2.1 void* ProcessInfo::scheduleData

Check specific scheduler docs for the contents of this pointer. Since the process struct remains static, this gives the ability for schedulers to store their own state without the kernel having to know ahead of time.

Referenced by cLottery::addProcess(), cMultiLevel::addProcess(), cMultiLevel::getNextToRun(), cFCFS::initProcScheduleInfo(), cRoundRobin::initProcScheduleInfo(),

cLottery::initProcScheduleInfo(), cMultiLevel::initProcScheduleInfo(), cFCFS::removeProcess(), cRoundRobin::removeProcess(), cLottery::removeProcess(), cMultiLevel::removeProcess(), cFCFS::setBlocked(), cRoundRobin::setBlocked(), cLottery::setBlocked(), cMultiLevel::setBlocked(), cFCFS::unblockProcess(), cRoundRobin::unblockProcess(), cLottery::unblockProcess(), and cMultiLevel::unblockProcess().

The documentation for this struct was generated from the following file:

- include/process.h

4.19 roundRobinInfo Struct Reference

Struct containing process info specific for Round-Robin scheduling.

`#include <round_robin.h>`

Public Attributes

- unsigned int blockedIndex

  Index position in blocked vector.

The documentation for this struct was generated from the following file:

- include/scheduler/round_robin.h

4.20 sMultiInfo Struct Reference

Struct containing process info specific for Multi-Level scheduling.

`#include <multi_level.h>`

Public Attributes

- int blockedIndex

  Index position in blocked vector.
- int level

  Current queue level.

The documentation for this struct was generated from the following file:

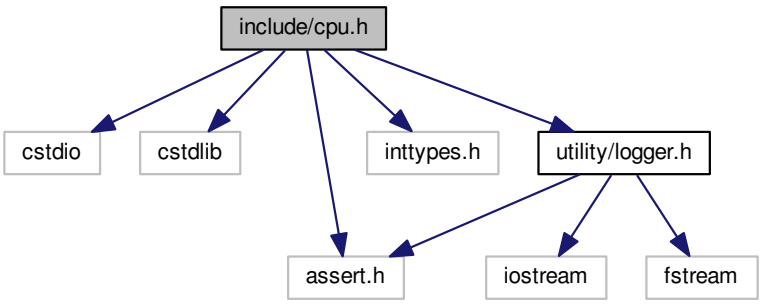
- include/scheduler/multi_level.h

Chapter 5

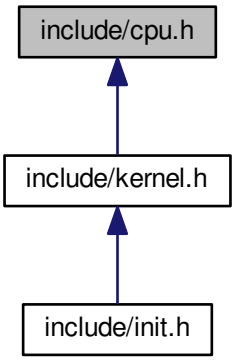
File Documentation

5.1 include/cpu.h File Reference

`#include <cstdio> #include <cstdlib> #include <assert.-h> #include <inttypes.h> #include "utility/logger.h"` Include dependency graph for cpu.h:



This graph shows which files directly or indirectly include this file:



Classes

- class cCPU

  A class for emulating a simple cpu.

Defines

- #define MAX_PARAMS 2

  Max number of execution parameters for any Opcode.
- #define MAX_PARAM_SIZE 256

  Maximum size in bytes for an execution parameter.

Typedefs

- typedef unsigned int **pidType**

Enumerations

- enum ePSW { PS_EXCEPTION = 0x1, PS_TERMINATE = PS_EXCEPTION << 1, PS_ABNORMAL = PS_TERMINATE << 1, PS_SYSCALL = PS_ABNORMAL << 1, PS_FINISHED = PS_SYSCALL << 1 }

  Enumeration of Program Status Word Flags.

5.1.1 Detailed Description

5.1.2 Define Documentation

5.1.2.1 #define MAX_PARAM_SIZE 256

Creates exception if exceeded.

5.1.3 Enumeration Type Documentation

5.1.3.1 enum ePSW

The program status word is a bit vector and this enumeration defines the meaning of particular bits. This is used in the interpretation of execution status.

Enumerator:

PS_EXCEPTION Executing process has created an exception.

PS_TERMINATE Executing process has finished.

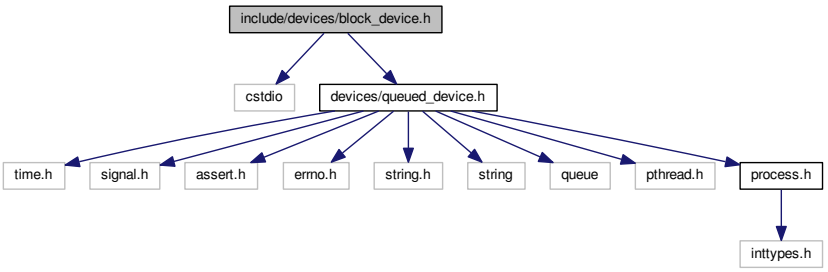
PS_ABNORMAL Process raised an exception by exiting abnormally. No 'E' instruction.

PS_SYSCALL Executing process has made a system call.

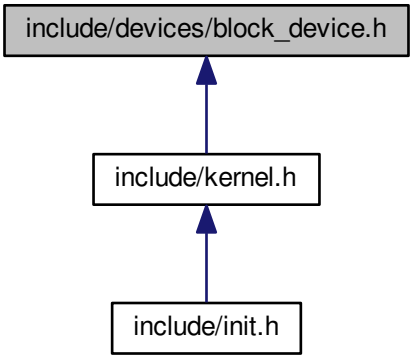
PS_FINISHED Executing process finished an instruction. No problems

5.2 include/devices/block_device.h File Reference

`#include <cstdio>` `#include "devices/queued_device.h"` Include dependency graph for block_device.h:



This graph shows which files directly or indirectly include this file:



Classes

- class cBlockDevice

  Queued block device.

Defines

- #define BLOCKSIG SIGRTMIN + 1

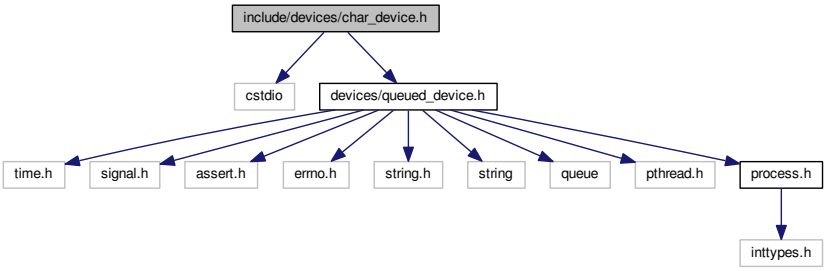
  Signal generated by BlockDevice.

5.2.1 Detailed Description

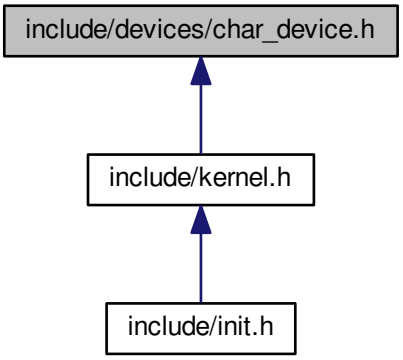
5.3 include/devices/char_device.h File Reference

`#include <cstdio>` `#include "devices/queued_device.h"` Include

dependency graph for char_device.h:



This graph shows which files directly or indirectly include this file:



Classes

- class cCharDevice

  Queued char device.

Defines

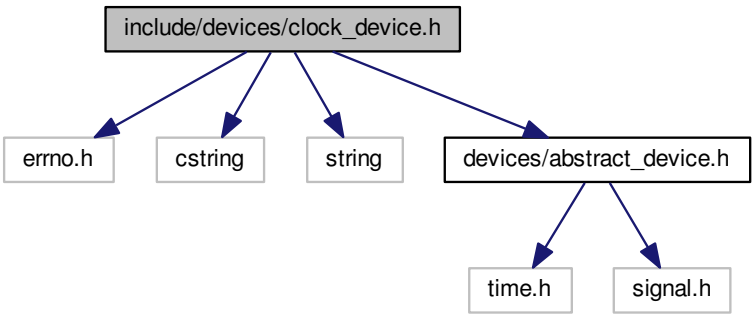
- #define CHARSIG SIGRTMIN + 2

  Signal generated by CharDevice.

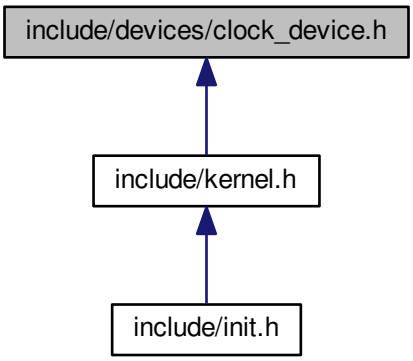
5.3.1 Detailed Description

5.4 include/devices/clock_device.h File Reference

`#include <errno.h>` `#include <cstring>` `#include <string>` × `#include "devices/abstract_device.h"` Include dependency graph for clock_device.h:



This graph shows which files directly or indirectly include this file:



Classes

- class ClockDevice

  Device for generating repeated clock interrupts.

Defines

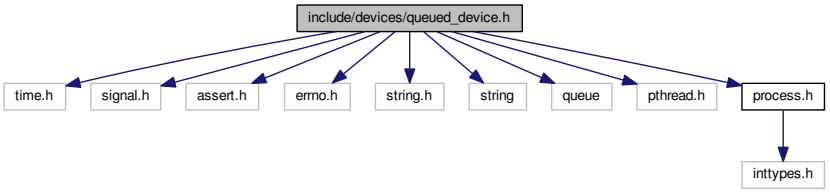
- #define **CLOCKID** CLOCK_REALTIME
- #define CLOCKSIG SIGRTMIN

  Signal generated by ClockDevice.

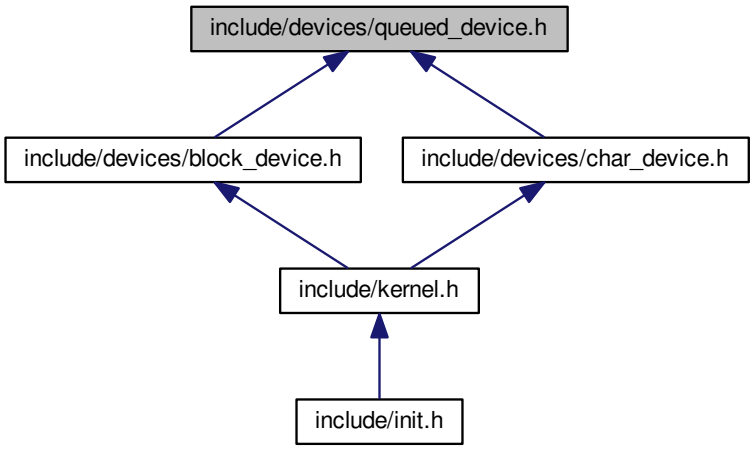
5.4.1 Detailed Description

5.5 include/devices/queued_device.h File Reference

`#include <time.h> #include <signal.h> #include <assert.h> #include <errno.h> #include <string.h> #include <string> #include <queue> #include <pthread.h> #include "process.h"` Include dependency graph for queued_device.h:



This graph shows which files directly or indirectly include this file:



Classes

- class cAbsQueuedDevice

  Abstract class for a device which queues requests.

Defines

- #define QD_CLOCKID CLOCK_REALTIME

  Type of clock used by these devices.
- #define USEC_IN_SEC 1000000

  Constant used in converting times.
- #define TIMEOUT_SCALE 5

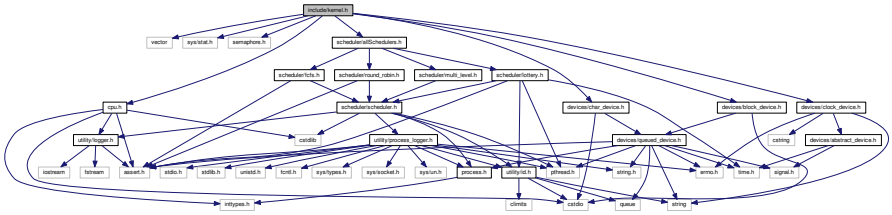
  In rare occurences the signal is never received so we set a timeout.

5.5.1 Detailed Description

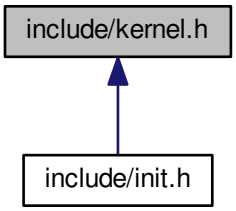
5.6 include/kernel.h File Reference

```
#include <vector> #include <sys/stat.h> #include <semaphore.-
h>   #include "cpu.h"   #include "devices/char_device.h" ×
#include "devices/block_device.h" #include "devices/clock-
```

_device.h" #include "scheduler/allSchedulers.h" Include dependency graph for kernel.h:



This graph shows which files directly or indirectly include this file:



Classes

- class cKernel

  Core managing class for this simulated OS.
- struct kernelError

  Struct containing kernel crash information.

Defines

- #define DEFAULT_TIMER 250000

  Default timer for clock interrupt.
- #define CDEVICE_SCALE 4

  Time scale for character devices relative to default clock.
- #define BDEVICE_SCALE 8

  Time scale for block devices relative to default clock.
- #define DEFAULT_CTIMER (DEFAULT_TIMER ∗ CDEVICE_SCALE)

Complete time for C Devices after scaling.

- #define DEFAULT_BTIMER (DEFAULT_TIMER * BDEVICE_SCALE)

  Complete time for B Devices after scaling.

- #define DEFAULT_PRIORITY 5

  Default priority assigned to newly created processes.

Variables

- static const char initProcessName [] = "main.trace"

  Name of the first program to run on the system.

- static const char traceLogFile [] = "trace.log"

  Name of trace log file.

- static const char procLogFile [] = "proc.log"

  Name of log file for process info.

5.6.1 Detailed Description

5.6.2 Define Documentation

5.6.2.1 #define DEFAULT_PRIORITY 5

Only used if no other priority is provided.

5.6.3 Variable Documentation

5.6.3.1 static const char initProcessName[] = "main.trace" [static]

When the kernel object is created, this program is loaded. It is run once cKernel::boot is called.

Referenced by cKernel::cKernel().

5.6.3.2 static const char procLogFile[] = "proc.log" [static]

This file is created by the process logger and it is where top gathers most of its process info.

See also

cProcessLogger

5.6.3.3 static const char traceLogFile[] = "trace.log" [static]

Per clocktick informatino is logged here.

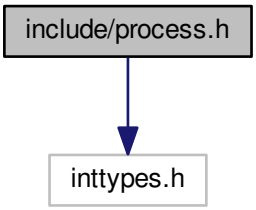
See also

initLog(const char* filename)

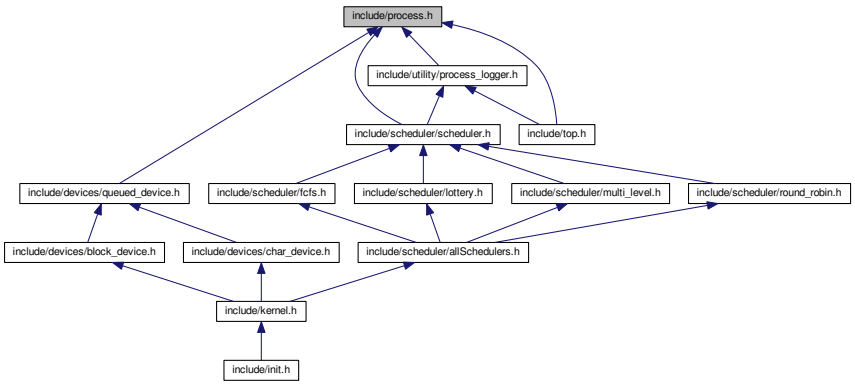
Referenced by cKernel::cKernel().

5.7 include/process.h File Reference

`#include <inttypes.h>` Include dependency graph for process.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct ProcessInfo

  Structure for containing process state and data.

Typedefs

- typedef unsigned int **pidType**

Enumerations

- enum eProcState { ready, running, blocked, terminated }

  Enumeration for process states.

5.7.1 Detailed Description

5.7.2 Enumeration Type Documentation

5.7.2.1 enum eProcState

Each values defines a current state and possible transitions.

Enumerator:

ready Process is ready to be run. Invariant State:

- Kernel has initialized it at some point
- Process should be preparred to run

Potential Transitions:

- running - Scheduler picks it to run next

running Process is currently running. A running process should implicilty be considered ready. The kernel may not notify the scheduler to transition the process to ready before asking for a a scheduling decision. It is acceptable for the scheduler to make a process ready without the kernel's consent when it is being asked for a scheduling decision, assuming it was previously running.

Invariant State:

- Process is on the cpu

Potential Transitions:

- ready - Scheduler picks someone else to run
- blocked - Makes blocking system call
- terminated - Causes exception in cpu or finished normally

blocked Process is blocked and cannot run. Invariant State:

- Process is blocked (for now it can only block on I/O)

Potential Transitions:

- ready - Kernel notifies scheduler that I/O has finished

terminated Process has been terminated. It will be cleaned up soon.

Invariant State:

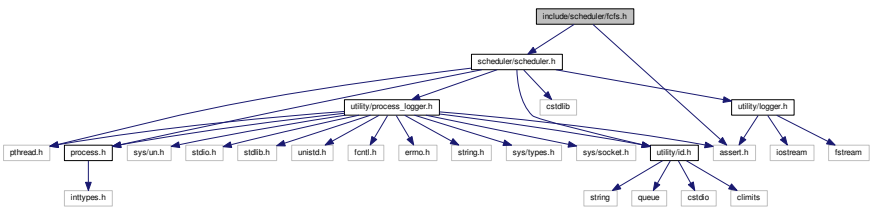
- Process either caused cpu exception or finished
- Process can no longer run

Potential Transitions:

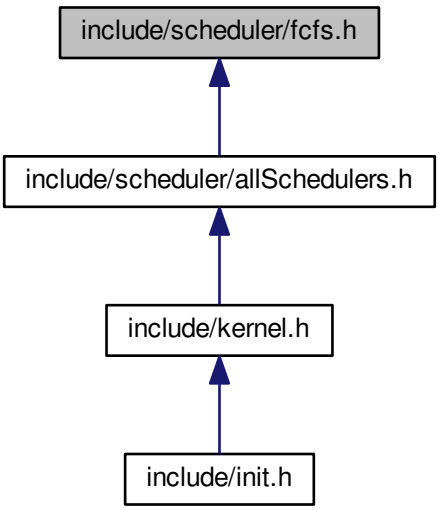
- None

5.8 include/scheduler/fcfs.h File Reference

`#include <assert.h>` `#include "scheduler/scheduler.h"` Include dependency graph for fcfs.h:



This graph shows which files directly or indirectly include this file:



Classes

- class cFCFS

  First-Come-First-Serve Scheduler.
- struct fcfsInfo

  Struct containing process info specific for FCFS scheduling.

Defines

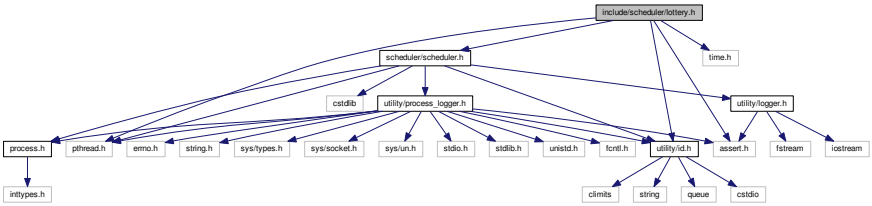
- #define **DEF_BLOCK_VEC_SIZE** 4

5.8.1 Detailed Description

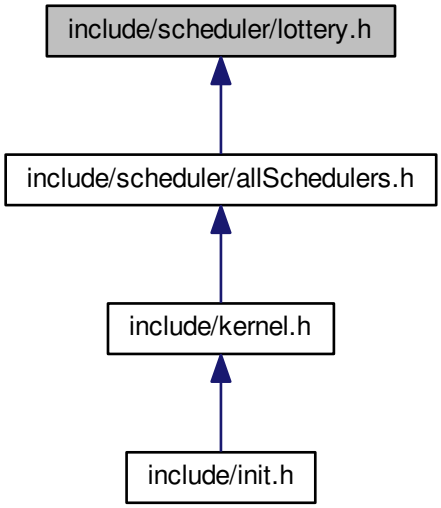
5.9 include/scheduler/lottery.h File Reference

```
#include <assert.h> #include <pthread.h> #include <time.-
h> #include "scheduler/scheduler.h" #include "utility/id.-
```

h" Include dependency graph for lottery.h:



This graph shows which files directly or indirectly include this file:



Classes

- class cLottery

  Lottery Scheduler.

- struct lotteryInfo

  Struct containing process info specific for Lottery scheduling.

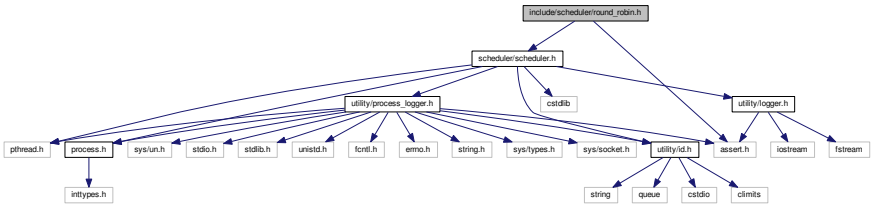
Defines

- #define **DEF_BLOCK_VEC_SIZE** 4

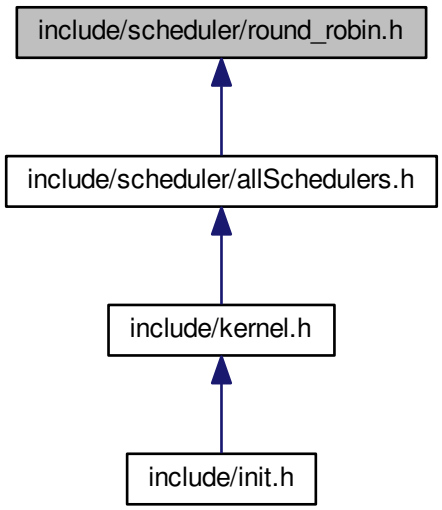
5.9.1 Detailed Description

5.10 include/scheduler/round_robin.h File Reference

`#include <assert.h>` `#include "scheduler/scheduler.h"` Include dependency graph for round_robin.h:



This graph shows which files directly or indirectly include this file:



Classes

- class cRoundRobin

  Round Robin Scheduler.

- struct roundRobinInfo

  Struct containing process info specific for Round-Robin scheduling.

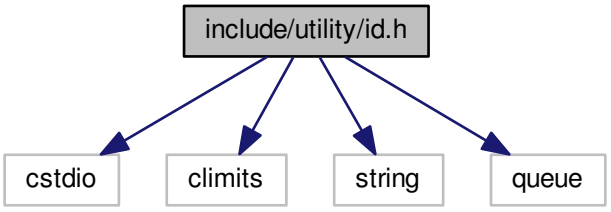
Defines

- #define **DEF_BLOCK_VEC_SIZE** 4
- #define **QUANTUM** 4

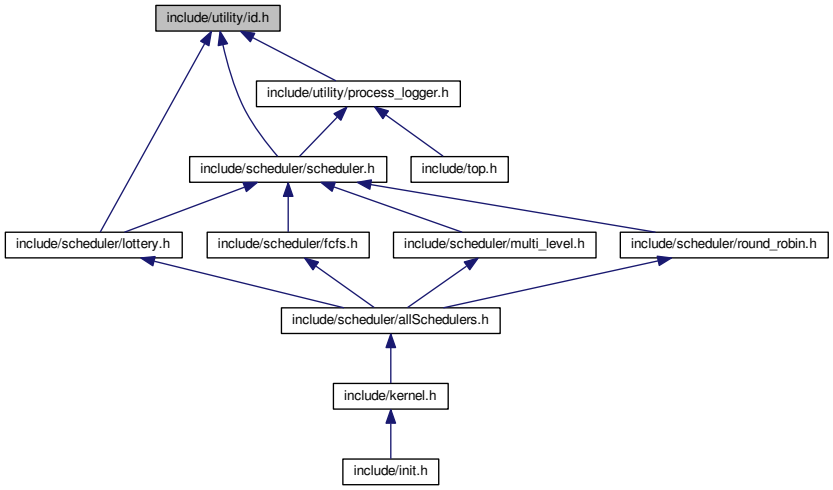
5.10.1 Detailed Description

5.11 include/utility/id.h File Reference

`#include <cstdio> #include <climits> #include <string>` ×
`#include <queue>` Include dependency graph for id.h:



This graph shows which files directly or indirectly include this file:



Classes

- class cIDManager

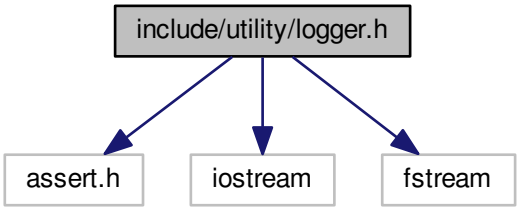
  A class for managing unique IDs.

5.11.1 Detailed Description

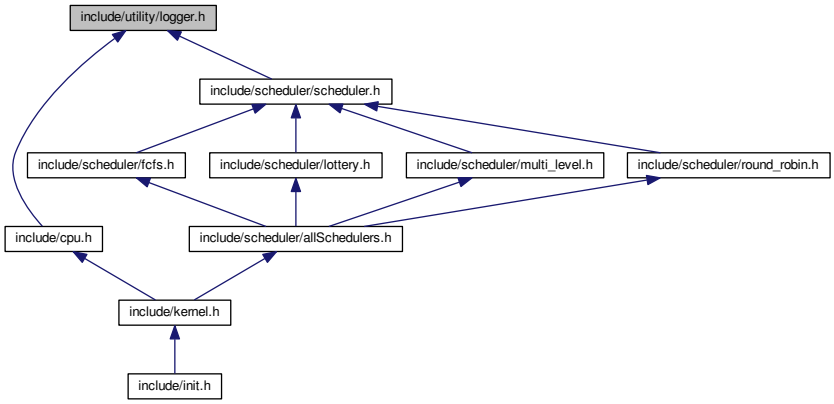
5.12 include/utility/logger.h File Reference

```
#include <assert.h> #include <iostream> #include <fstream> ×
```

Include dependency graph for logger.h:



This graph shows which files directly or indirectly include this file:



Functions

- FILE * initLog (const char *filename)

  Initialize a trace log at filename.

- void closeLog ()

  Close the file stream for the trace log.

- FILE * getStream ()

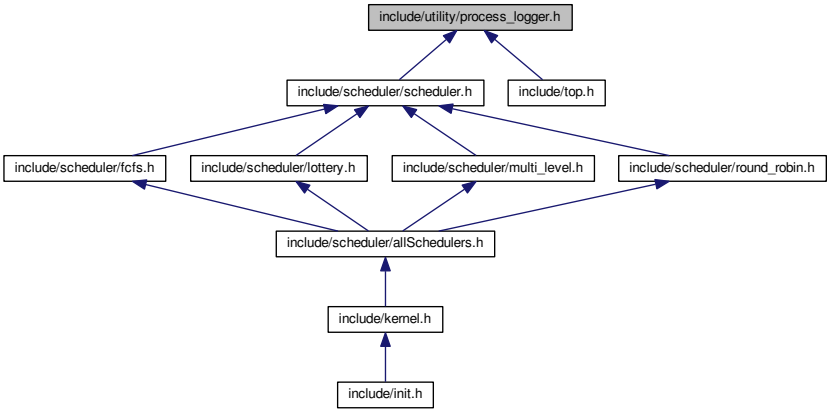
  Get the file stream to write to.

5.12.1 Detailed Description

5.13 include/utility/process_logger.h File Reference

`#include <stdio.h>` `#include <stdlib.h>` `#include <unistd.h>` `#include <fcntl.h>` `#include <assert.h>` `#include <errno.h>` `#include <string.h>` `#include <pthread.h>` `#include <sys/types.h>` `#include <sys/socket.h>` `#include <sys/un.h>` `#include "process.h"` `#include "utility/id.h"` Include dependency graph for process_logger.h:

This graph shows which files directly or indirectly include this file:



Classes

- class cProcessLogger

  Class specifically for logging process state information.

Defines

- #define MAX_LINE_LENGTH 45

Max line length for a process entry in the log file.

Enumerations

- enum **pivotType** { **pivotMiddle**, **pivotRandom** }

Functions

- template<typename T >
  void **QuickPartition** (std::vector< T > items, int(*sort_fn)(T, T), int left, int right, int pivot=pivotMiddle)
- template<typename T >
  void **QuicksortVector** (std::vector< T > items, int(*sort_fn)(T, T), pivotType p-Type, int start, int end)

Variables

- static const char procNameReq [] = "proc.log.req"
  Name of unix socket file for processes like top to request process names.
- static const char outputFormat [] = "%u %d %d %d %d"
  Format for process info in the log file.
- static const char requestError [] = "INVALID_ID"
  Return value when a invalid process ID was requested.

5.13.1 Detailed Description

5.13.2 Variable Documentation

5.13.2.1 static const char outputFormat[] = "%u %d %d %d %d" [static]

Scanf is used to print the information in this format to a buffer. Then this buffer is padded to fill MAX_LINE_LENGTH and then it is written out to the appropriate line in the file.

5.13.2.2 static const char procNameReq[] = "proc.log.req" [static]

Since process trace names are variable, their names are stored in a string objects and then served upon request to this socket. While filenames do have a max size, it is more efficient to do it this way when you consider that most trace file names will not be near the max.

5.13.2.3 static const char requestError[] = "INVALID_ID" [static]

If another process requests an invalid ID on the request socket, this is the corresponding message.