Priority Inversion

When a low-priority task blocks a higher-priority one, a priority inversion is said to occur

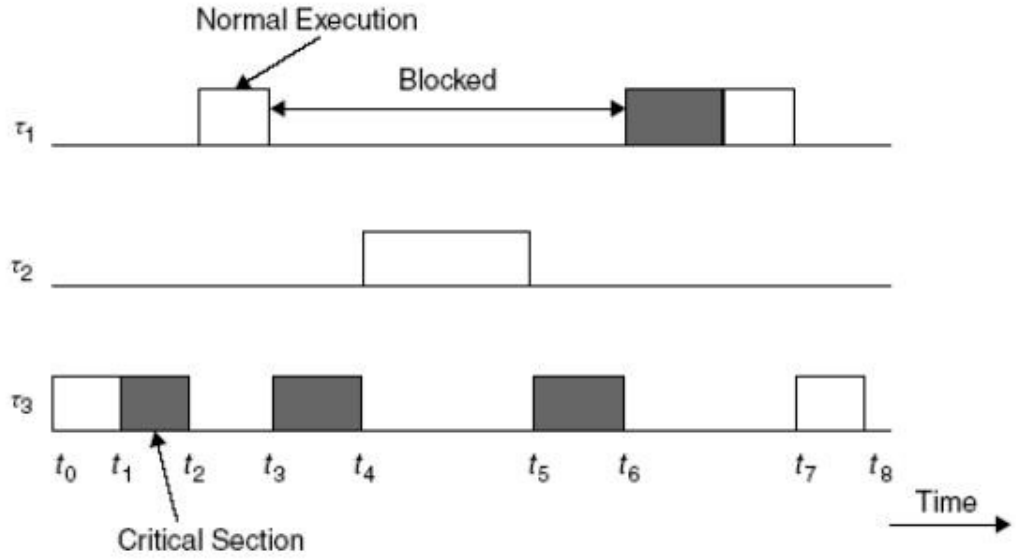


Figure 3.16 A priority inversion problem.

Assume that priorities: p1>p2>p3, and tasks 1 and 3 share the same critical resource

Priority Inheritance Protocol

In the PIP the priorities of tasks are dynamically changed so that the priority of any task in a critical section gets the priority of the highest task pending on that same critical resource.

- The highest-priority task, τ, relinquishes the processor whenever it seeks to lock the semaphore guarding a critical section that is already locked by some other job.
- If a task, τ_1, is blocked by τ_2 and $\tau_1 \succ \tau_2$, (i.e., τ_1 has precedence over τ_2), task τ_2 inherits the priority of τ_1 as long as it blocks τ_1. When τ_2 exits the critical section that caused the block, it reverts to the priority it had when it entered that section.
- Priority inheritance is transitive. If τ_3 blocks τ_2, which blocks τ_1 (with $\tau_1 \succ \tau_2 \succ \tau_3$), then τ_3 inherits the priority of τ_1 via τ_2.

Priority Inheritance Protocol

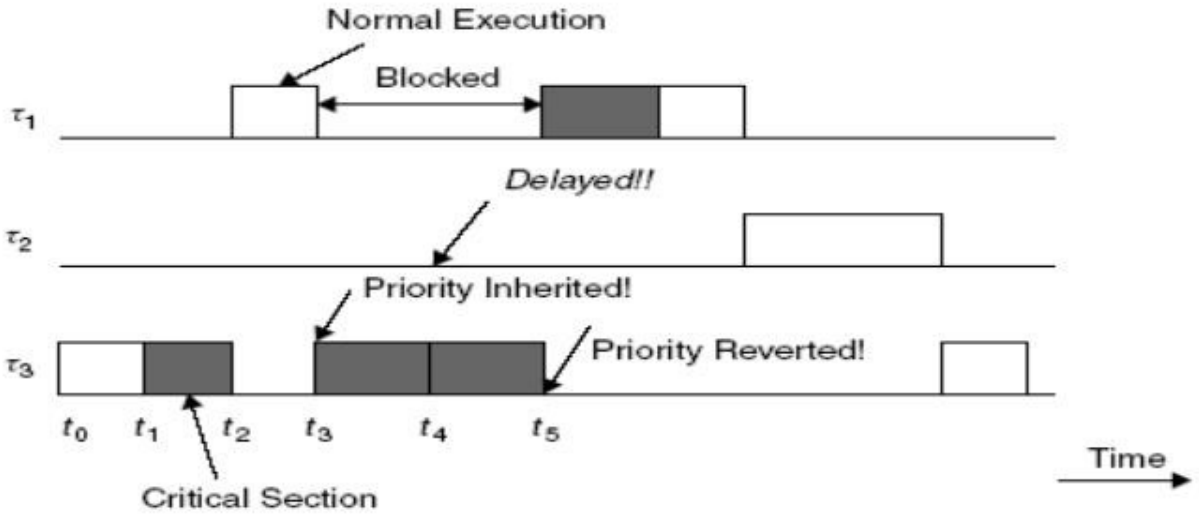


Figure 3.17 Priority Inheritance Protocol illustration.

It is important to point out that the Priority Inheritance Protocol does not prevent deadlock. In fact, Priority Inheritance can cause deadlock or multiple blocking. Nor can it prevent other problems induced by semaphores. For example, consider the following sequence (with $\tau_1 \succ \tau_2$):

τ_1: Lock S_1; Lock S_2; Unlock S_2; Unlock S_1
τ_2: Lock S_2; Lock S_1; Unlock S_1; Unlock S_2

Priority Inheritance Protocol

Finally, a notorious incident of the priority inversion problem occurred in 1997 in NASA's Mars Pathfinder Space mission's *Sojourner* rover vehicle, which was used to explore the surface of Mars. In this case the Mil-std-1553B information bus manager was synchronized with mutexes. Accordingly a meteorological data-gathering task that was of low priority and low frequency blocked a communications task that was of higher priority and higher frequency. This infrequent scenario caused the system to reset. The problem would have been avoided if the priority inheritance mechanism provided by the commercial real-time operating system (just mentioned) had been used. But it had been disabled. Fortunately, the problem was diagnosed in ground-based testing and remotely corrected by reenabling the priority inheritance mechanism

Priority Ceiling Protocol

Each resource is assigned a priority ceiling equal to the priority of the highest priority task that can use it

Table 3.9 Data for the Priority Ceiling Protocol illustration

Critical Section	Accessed by	Priority Ceiling
S_1	τ_1, τ_2	$P(\tau_1)$
S_2	τ_2, τ_3	$P(\tau_2)$

Priority Ceiling Protocol

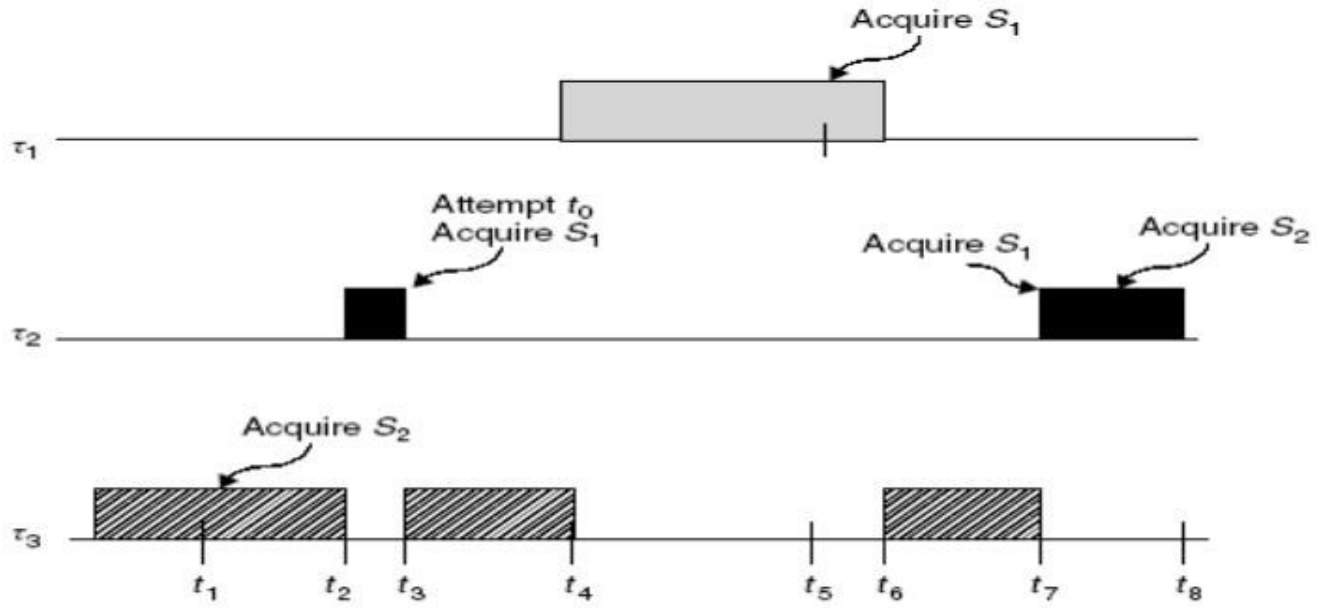


Figure 3.18 Illustration of the Priority Ceiling Protocol in action.

τ_1: Lock S_1; Unlock S_1

τ_2: Lock S_1; Lock S_2; Unlock S_2; Unlock S_1

τ_3: Lock S_2; Unlock S_2

Priority Ceiling Protocol

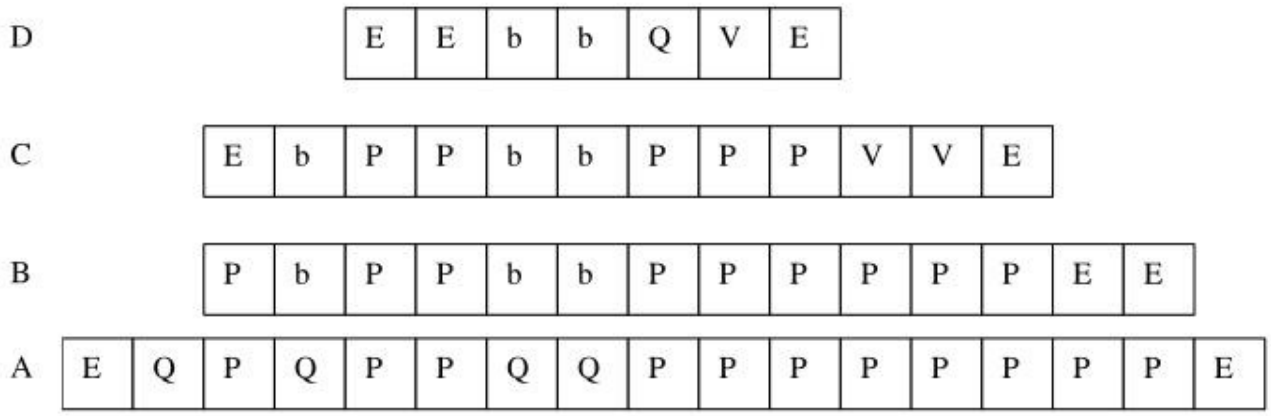
Suppose that τ_3 starts executing first, locks the semaphore S_2 at time t_1 and enters the critical section. At time t_2, τ_2 starts executing, preempts τ_3, and attempts to lock semaphore S_1 at time t_3. At this time, τ_2 is suspended because its priority is not higher than priority ceiling of semaphore S_2, currently locked by τ_3. Task τ_3 temporarily inherits the priority of τ_2 and resumes execution. At time t_4, τ_1 enters, preempts τ_3, and executes until time t_5, where it tries to lock S_1. Note that τ_1 is allowed to lock S_1 at time t_5, as its priority is greater than the priority ceiling of all the semaphores currently being locked (in this case, it is compared with S_2). Task τ_1 completes its execution at t_6 and lets τ_3 execute to completion at t_7. Task τ_3 is then allowed to lock S_1, and subsequently S_2, and completes at t_8.

ORIGINAL CEILING PRIORITY PROTOCOL (OCPP)

- Each process has a static default priority assigned (for example, by deadline monotonic scheme: less deadline – greater priority, or by rate monotonic scheme: less period – greater priority).
- Each resource has a static ceiling value defined, this is the maximum priority of the processes that use it (assume that greater priority is represented by greater value).
- A process has a dynamic priority which is the maximum of its own static priority and any it inherits due to it blocking higher-priority processes (when process blocks higher-priority task it gets temporarily this higher priority).
- A process can only lock a resource if its dynamic priority is higher than the ceiling of any currently locked resource (excluding any that it has already locked).

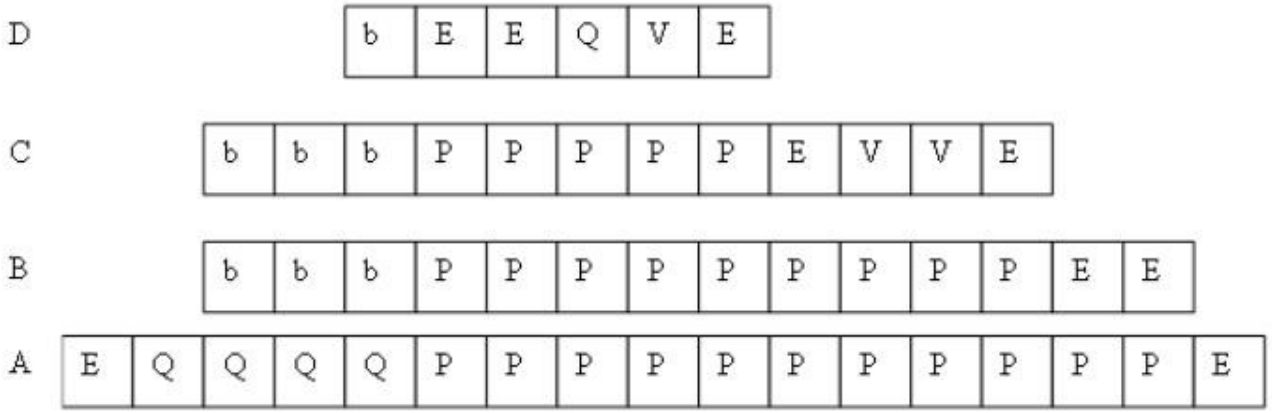
Original Ceiling Priority Protocol

Process	Priority	Execution sequence	Release time
A	1	EQQQQE	0
B	2	EE	2
C	3	EVVE	2
D	4	EEQVE	4



IMMEDIATE CEILING PRIORITY PROTOCOL (ICPP)

1. Each process has a static default priority assigned.
2. Each resource has a static ceiling value defined, this is the maximum priority of the processes that use it.
3. A process has a dynamic priority that is the maximum of its own static priority and the ceiling values of any resources it has locked.



Memory Management

Stack management:

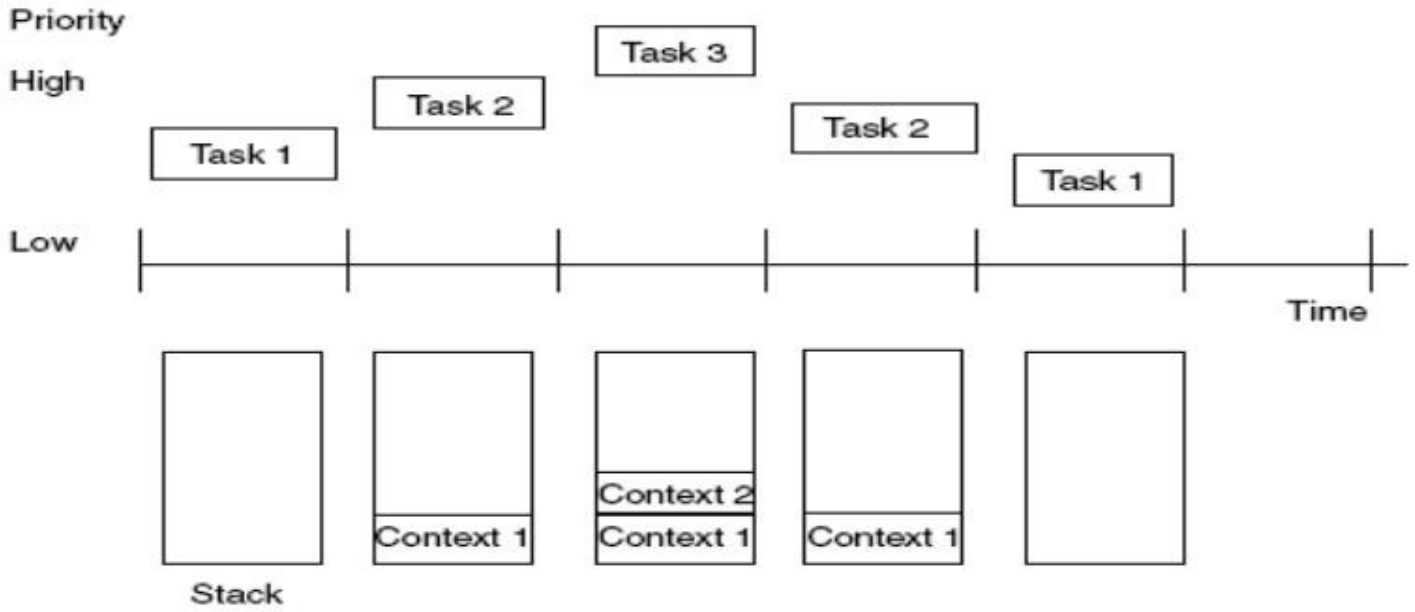


Figure 3.19 Activity on the run-time stack as tasks are interrupted.

Ideally, provision of at least one more task than anticipated should be allocated to the stack to allow for spurious interrupts and time overloading

Memory Management

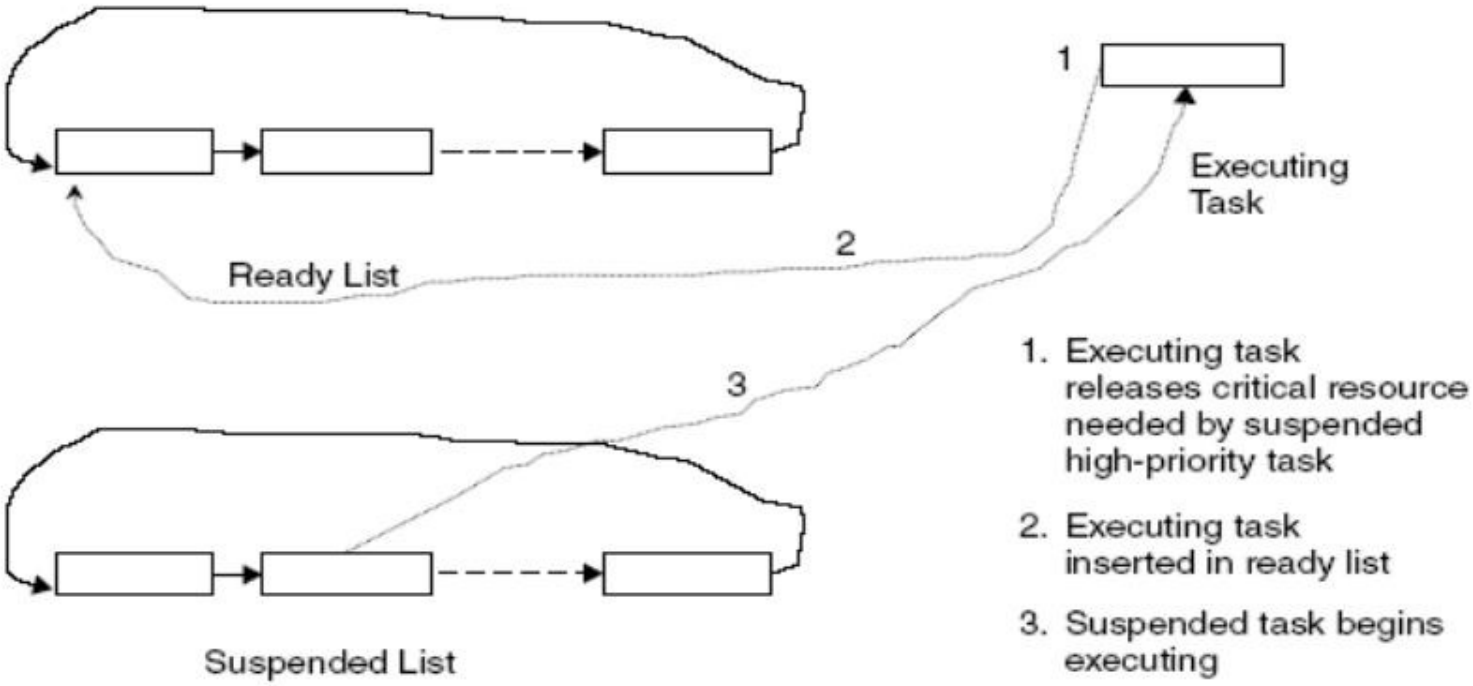


Figure 3.21 Memory management in the task-control-block model.

Swapping

A process is saved to an external device when its time expires. The next process is loaded instead

Overlays

Main(){

a(); b(); c()

}

This program can be represented as a root segment always resident in memory, and three overlay segments a, b, c

Memory required in this case is

max(mem_a,mem_b,mem_c)

instead of

mem_a+mem_b+mem_c

Partitions

Static partitions with absolute addresses

Number of partitions and the sizes are fixed. Tasks are compiled for the particular partition. There may be a skew in partitions loading

Static partitions with relative addresses

Number of partitions and the sizes are fixed. Tasks are compiled for any partition, need adjustment.

Happens internal fragmentation

Dynamic partitions

Number of partitions vary, tasks get partition of the necessary size. External fragmentation can occur => garbage collection (de-fragmentation)

De-fragmentation

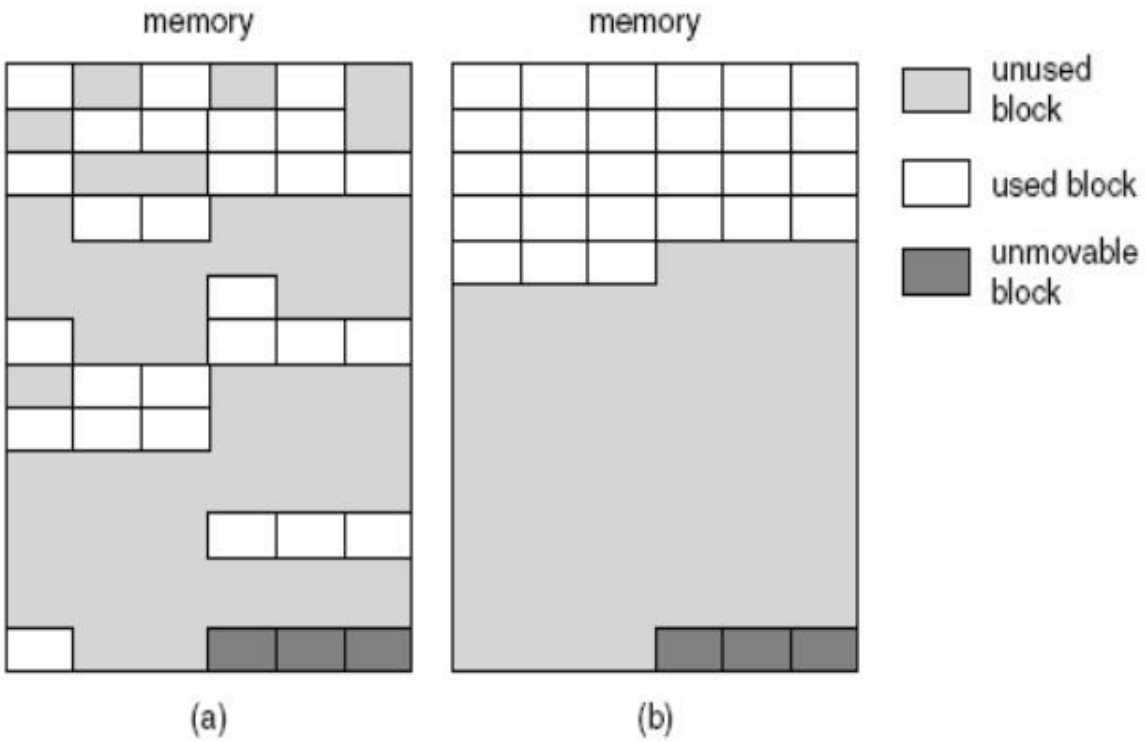


Figure 3.22 Fragmented memory (a) before and (b) after compaction, including unmovable blocks (for example, representing the operating system root program).

Virtual Memory

Segmented: virtual address = (segment, offset)

Address=start(segment)+offset

Paged: (page, offset)

Address=start(page)+offset

Segmented-paged: (segment, page, offset)

Address=start(segment)+start(page)+offset

Replacement algorithms

FIFO, LRU (Least recently used)

If we have 4 page memory and

2 4 6 8 is a a page reference string (page 2 is the least and page 8 is the most recently used) then

2 4 6 8 9 2 4 6 8 9 2 4 6 8 9 will lead to page thrashing

Memory locking, Working sets

I/O Problems

Disk fragmentation

Contiguous file allocation: place parts of a file in adjacent sectors

Elevator algorithm: reorder queries so that they will be served with minimal magnetic head replacement

Criteria for RTOS

1. Interrupt latency
2. Number of processes system can maintain
3. Memory for OS
4. Scheduling mechanisms set (round-robin, preemptive, etc)
5. Available communication methods (mutexes, semaphores, etc)
6. After sale support

RTOS Criteria

7. Software development kits available
8. Hardware platforms support
9. Source code availability
10. Task context switch time
11. RTOS cost
12. Compatibility with other RTOS
13. Networks and network protocols supported by RTOS

A fitness metric can be

$$M = \sum_{i=1}^{13} \omega_i m_i$$

Table 3.10 Summary data for real-time operating system A[a]

Criterion	Description	Rating	Comment
m_1	Minimum interrupt latency	*	CPU dependent
m_2	Number of tasks supported	0.5	32-Thread priority levels
m_3	Memory requirements	0.7	ROM: 60 K
m_4	Scheduling mechanism	0.25	Preemptive
m_5	Intertask synchronization mechanism	0.5	Direct message passing
m_6	Software support (warranty)	0.5	Paid phone support
m_7	Software support (compiler)	1	Various
m_8	Hardware compatibility	0.8	Various
m_9	Royalty free	0	No
m_{10}	Source available	1	Yes
m_{11}	Context switch time	*	NA
m_{12}	Cost	0.7	Approximately $2500
m_{13}	Supported network protocols	1	Various

[a]Some of the specific details have been deliberately omitted to preserve the identity of the product.

Table 3.11 Decision table for inertial measurement system

Criterion	Description	Weight, w_1	A	B	C	D	E
m_1	Minimum interrupt latency	1	0.5	0.8	1	0.5	1
m_2	Number of tasks supported	0.1	0.5	0.5	0.5	1	1
m_3	Memory requirements	1	0.7	0.2	0.5	1	0.9
m_4	Scheduling mechanism	0.5	0.25	0.5	0.25	1	0.25
m_5	Intertask synchronization mechanism	1	0.5	1	0.5	1	1
m_6	Software support (warranty)	1	0.5	0.5	1	0.8	1
m_7	Software support (compiler)	1	1	0.75	1	1	0.5
m_8	Hardware compatibility	0.1	0.8	0.5	0.2	1	0.2
m_9	Royalty free		0	1	1	1	1
m_{10}	Source available	1	1	1	0	0.4	1
m_{11}	Context switch time	1	0.5	0.5	0.5	1	0.5
m_{12}	Cost	0.4	0.5	0.5	0.1	0.1	0.7
m_{13}	Supported network protocols	0.5	1	1	1	1	0.6
M			5.66	5.80	5.24	6.94	6.73

Here the metric M in equation 3.10 is computed for five candidate RTOS (A through E). From the last row it can be seen that RTOS D is the best fit in this case.

POSIX (Portable Operating System Interface for Computer Environments, IEEE Standard)

http://www.unix.org/single_unix_specification/

```
/** POSIX Condition Variables **/
/* Creating/Destroying Condition Variables */
    pthread_cond_init(condition, attr)
    pthread_cond_destroy(condition)
    pthread_condattr_init(attr)
    pthread_condattr_destroy(attr)

/* Waiting/Signalling On Condition Variables */
    pthread_cond_wait(condition, mutex)
    pthread_cond_signal(condition)
    pthread_cond_broadcast(condition)
```

It is essential to associate a mutex with a condition variable. A thread locks the mutex for some shared data and then checks whether data are in the proper state. If no, the thread waits on the appropriate condition variable. Waiting on the condition variable automatically unlocks the mutex. When some thread puts the data in the proper state, it wake a waiting thread by signaling. One thread comes out of its wait state with the mutex locked

POSIX Semaphores

Binary and counting

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
/* Initializes the semaphore object pointed by 'sem' */

int sem_destroy(sem_t *sem);
/* Destroys a semaphore object and frees up the resources it might hold */

/* The following three functions are used in conjunction with other
   processes. See man pages for more details.
*/
sem_t *sem_open(const char *name, int oflag, ...);
int sem_close(sem_t *sem);
int sem_unlink(const char *name);

int sem_wait(sem_t *sem);
/* Suspends the calling thread until the semaphore pointed to by 'sem' has
   non-zero count. Decreases the semaphore count. */

int sem_trywait(sem_t *sem);
/* A non-blocking variant of sem_wait. */

int sem_post(sem_t *sem);
/* Increases the count of the semaphore pointed to by 'sem'. */

int sem_getvalue(sem_t *sem, int *sval);
/* Stores the current count of the semaphore 'sem' in 'sval'. */
```

POSIX Messages

Message queues work by exchanging data in buffers. Any number of processes can communicate through message queues. Message notification can be synchronous or asynchronous. The POSIX message passing through message-queue facilities provide a deterministic, efficient means for interprocess communication. Real-time message passing is designed to work with shared memory in order to accommodate the needs of real-time applications with an efficient, deterministic mechanism to pass arbitrary amounts of data between cooperating processes. The following prototypes describe the POSIX messaging capabilities.

```
mqd_t mq_open(const char *name, int oflag, ...);
/* Connects to, and optionally creates, a named message queue. */

int mq_send(mqd_t mqdes, const char *msg_ptr, oskit_size_t msg_len,
            unsigned int msg_prio);
/* Places a message in the queue. */

int mq_receive(mqd_t mqdes, char *msg_ptr, oskit_size_t msg_len,
               unsigned int *msg_prio);
/* Receives (removes) the oldest, highest priority message from the queue. */

int mq_close(mqd_t mqdes);
/* Ends the connection to an open message queue. */

int mq_unlink(const char *name);
/* Ends the connection to an open message queue and causes the
 queue to be removed when the last process closes it. */

int mq_setattr(mqd_t mqdes, const struct mq_attr *mqstat, struct mq_attr
*omqstat);
int mq_getattr(mqd_t mqdes, struct mq_attr *mqstat);
/* Set or get message queue attributes. */

int mq_notify(mqd_t mqdes, const struct sigevent *notification);
/* Notifies a process or thread that a message is available in the queue. */
```

Real-Time POSIX Signals

Signals are software representation of interrupts or exception occurrences. No routine should be called from a signal handler that might cause the handler to block.

Signals are used for

- Exception handling
- Process notification of asynchronous event occurrence
- Process termination in an abnormal situation
- Interprocess communication

Real-Time POSIX Signals

However, there are several limitations of standard POSIX signals on their use in real-time applications. These include:

- Lack of signal queueing
- No signal delivery order
- Poor information content

POSIX real-time extensions (POSIX.4) improves the POSIX signals to applications. POSIX.4 defines a new set of application-defined real-time signals, and these signals are numbered from `SIGRTMIN` to `SIGRTMAX`. There must be `RTSIG_MAX >8` signals in between these two limits.

The `sigaction` defines all the details that a process need to know when a signal arrives. As real-time signals can be queued, the queueing option for a real-time signal is chosen by setting bit `SA_SIGINFO` in the `sa_flags` field of the `sigaction` structure of the signal.

Real-Time POSIX Signals

```
struct sigaction{
  void (*sa_handler)();
  sigset_t sa_mask;
  int sa_flags;//SA_NOCLDSTOP or SA_SIGINFO
  void (*sa_sigaction)(int, siginfo_t*, void*);
  /*used for real-time signals!! Also, ''SA_SIGINFO''
    is set in ''sa_flags.''
   */
};

int sigaction(int sig, const struct sigaction *reaction,
              struct sigaction *oldreaction);
```

Real-time signals can carry extra data. `SA_SIGINFO` increases the amount of information delivered by each signal. If `SA_SIGINFO` is set, then the signal handlers have as an additional parameter a pointer to a data structure called a `siginfo_t` that contains the date value to be piggybacked.

The `sigqueue()` includes an application-specified value (of type `sigval`) that is sent as a part of the signal. It enables the queuing of multiple signals for any task. Real-time signals can be specified as offsets from `SIGRTMIN`. All signals delivered with `sigqueue()` are queued by numeric order, lowest numbered signals delivered first.

POSIX.4 provides a new and more responsive (or fast) synchronous signal-wait function called `sigwaitinfo`. Upon arrival of the signal, it does not call the signal handler (unlike `sigsuspend`), but unblocks the calling process.

System POSIX Signals

Signal	Default Action	Description
SIGABRT	A	Process abort signal.
SIGALRM	T	Alarm clock.
SIGBUS	A	Access to an undefined portion of a memory object.
SIGCHLD	I	Child process terminated, stopped,
[XSI] ⌦		or continued. ⌫
SIGCONT	C	Continue executing, if stopped.
SIGFPE	A	Erroneous arithmetic operation.
SIGHUP	T	Hangup.
SIGILL	A	Illegal instruction.
SIGINT	T	Terminal interrupt signal.
SIGKILL	T	Kill (cannot be caught or ignored).
SIGPIPE	T	Write on a pipe with no one to read it.
SIGQUIT	A	Terminal quit signal.
SIGSEGV	A	Invalid memory reference.
SIGSTOP	S	Stop executing (cannot be caught or ignored).
SIGTERM	T	Termination signal.

T Abnormal termination of the process. The process is terminated with all the consequences of *_exit()* except that the status made available to *wait()* and *waitpid()* indicates abnormal termination by the specified signal.

A Abnormal termination of the process.
[XSI] ⌦ Additionally, implementation-defined abnormal termination actions, such as creation of a **core** file, may occur. ⌫

I Ignore the signal.

S Stop the process.

C Continue the process, if it is stopped; otherwise, ignore the signal.

Clocks and Timers

Typically the system time is represented by a 32 bit unsigned integer, while tick value (time resolution) is represented by a float-point value

Table 3.13 System lifetime

Tick	Lifetime
1 microsecond	71.6 minutes
1 millisecond	50 days
10 millisecond	16 months
1 second	136 years

If *k* denotes system tick, and *n* is the value stored in sys_clock, then the actual time elapsed is *kn*

POSIX Clocks

POSIX allows many clocks to be used by an application. Each clock has its own identifier of type clockid_t. Commonly supported and used identifier is CLOCK_REALTIME that is systemwide clock visible for all processes.

Table 3.14 POSIX clock functions

Function	Description
`clock_getres`	Returns the resolution of the specified clock `int clock_getres(clockid_t clock_id, struct timespec *res)`
`Clock_gettime`	Returns the current value for the specified value
	`int clock_gettime(clockid_t clock_id, struct timespec *tp)`
`Clock_settime`	Sets the specified clock to the specified value `int clock_settime(clockid_t clock_id, const struct timespec *tp)`

Data structure timespec has two fields of the long integer type representing the value in number of seconds since the Epoch (tv_sec) and in nano-seconds (tv_nsec). Epoch started 00:00:00 1.1.1970

POSIX Clocks

3.5.7.4 Retrieving System Time The `clock_gettime` function returns the value of the systemwide clock as the number of elapsed seconds since the Epoch. The `timespec` data structure (used for the `clock_gettime` function) also contains a member to hold the value of the number of elapsed nanoseconds not comprising a full second.

```
#include <unistd.h>
#include <time.h>
main(){
 struct timespec ts;
 clock_gettime(CLOCK_REALTIME, &ts);
 printf(''clock_gettime returns:\n'');
 printf(''%d seconds and %ld nanoseconds\n'', ts.tv_sec, ts.tv_nsec); }
```

3.5.7.5 System Clock Resolution The system clock resolution on DEC's Alpha system is 1/1024 seconds or 976 microseconds),[7] that is, the system maintains time by adding 976 microseconds at every clock interrupt. The actual time period between clock ticks is exactly 976.5625 microseconds. The missing 576 microseconds (1024 * 0.5625) are added at the end of the 1024th tick, that is the 1024th tick advances the system time by 1552 microseconds.

Note that if an application program requests a timer value that is not an exact multiple of the system clock resolution (an exact multiple of 976.5625 microseconds), the actual time period counted down by the system will be slightly larger than the requested time period. A program that asks for a periodic timer of 50 milliseconds will actually get a time period of 50.78 milliseconds.

POSIX Timers

```
#include<signal.h>
#include<time.h>
timer_t timer_create(clockid_t clock_id, struct sigevent *event,
                     timer_t *timer_id);
```

As per POSIX standard, different platforms can have multiple time bases, but every platform must support at least the `CLOCK_REALTIME` time base. A timer based upon the system clock called `CLOCK_REALTIME` can be created. The `seconf` argument `event` points to a structure that contains all the information needed concerning the signal to be generated. This is essentially used to inform the kernel about what kind of event the timer should deliver whenever it "fires." By setting it `NULL`, the system is forced to use default delivery, which is defined to be `SIGALRM`.

The return value from `timer_create()` is effectively a small integer that just acts as an index into the kernel's timer tables.

POSIX Timers

```
struct itimerspec{
      struct timespec it_value,
                      it_interval;
};
int timer_settime (timer_t timerid, int flag,
                   struct itimerspec *value,
                   struct itimerspec *oldvalue);
```

This function sets the next expiration time for the timer specified. If flag is set to `Timer_ABSTIME`, then the timer will expire when the clock reaches the absolute value specified by `*value.it_value`. If flag is not set to `TIMER_ABSTIME`, the timer will expire when the interval specified by `value->it_value` passes. If `*value.it_interval` is nonzero, then a periodic timer will go off every `value->it_interval` after `value->it_value` has expired. Any previous timer setting is returned in `*oldvalue`. For example, to specify a timer that executes only once, 10.5 seconds from now, specify the following values for the members of the `itimerspec` structure:

```
newtimer_setting.it_value.tv_sec = 10;
newtimer_setting.it_value.tv_nsec = 500000000;
newtimer_setting.it_interval.tv_sec = 0;
newtimer_setting.it_interval.tv_nsec = 0;
```

To arm a timer to execute 15 seconds from now and then at 0.25-second intervals, specify the following values:

```
newtimer_setting.it_value.tv_sec = 15;
newtimer_setting.it_value.tv_nsec = 0;
newtimer_setting.it_interval.tv_sec = 0;
newtimer_setting.it_interval.tv_nsec = 250000000;
```

POSIX Asynchronous Input/Output

AIO operations use AIO Control Block aiocb:

```
struct aiocb{
  int              aio_fildes;     // File descriptor
  off_t            aio_offset;     // File offset
  volatile void    *aio_buf;       // Pointer to buffer
  size_t           aio_nbytes;     // Number of bytes to transfer

  int              aio_reqprio;    // Request priority offset
  struct sigevent aio_sigevent;    // Signal structure
  int              aio_lio_opcode;// Specifies type of I/O operation
};
```

It is important to understand what actually happens when the `aio_read/ aio_write(...)` functions are called. In fact, the following code is performed:

```
lseek(a.aio_fildes, ...); // Seek to position
read(a.aio_fildes,...);   // Read data
sigqueue(...);            // Queue a signal to a process
```

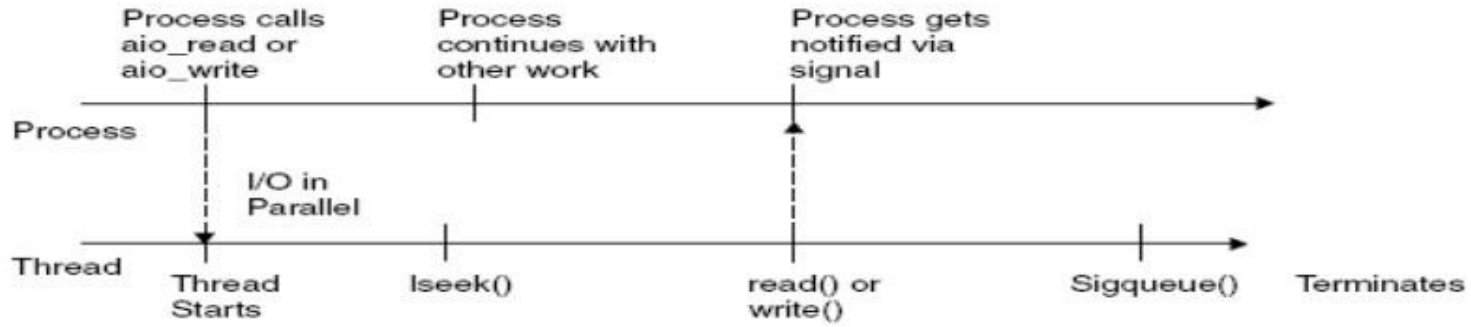


Figure 3.23 Asynchronous I/O operation.

POSIX Memory Locking

POSIX allows for a simple procedure to lock the entire process down.

```
#include <unistd.h>
#ifdef _POSIX_MEMLOCK
#include<sys/mman.h>
int mlockall(int flags);
int munlockall(void);
```

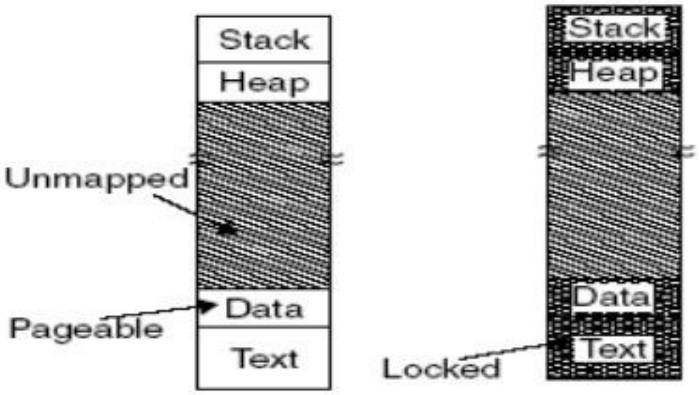


Figure 3.24 `mlockall` operation.

POSIX permits the user to lock down part of the process:

```
#include <unistd.h>
#ifdef _POSIX_MEMLOCK_RANGE
#include<sys/mman.h>
int mlock(void *address, size_t length);
int munlock(void *address, size_t length);
#endif
```

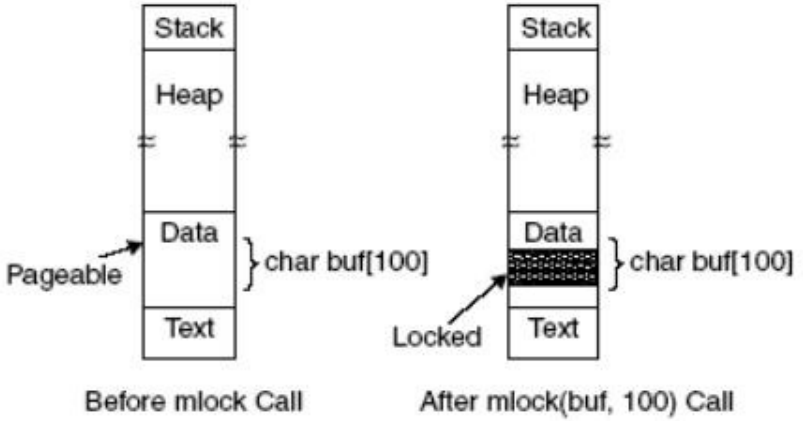


Figure 3.25 `mlock` operation.