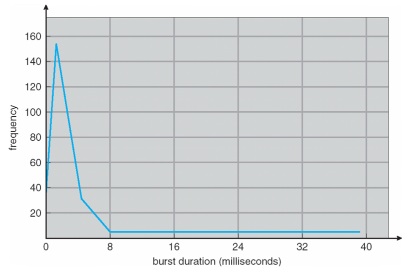
Operating Systems

Scheduling
(and Context Switching)

Scheduling

- Scheduling is a critical topic in Operating Systems
- A key observation is that processes tend to use the CPU in bursts
  - Period of CPU activity followed by waiting on I/O
- Xinu is priority-based but there are many other possibilities

Histogram of CPU-burst Times



Silberschatz, Galvin, and Gagne "Operating System Concepts, Ninth Edition ", Chapter 6

Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced

Optimization Criteria

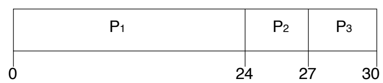
- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

5

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1, P_2, P_3
  The Gantt Chart for the schedule is:



- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$

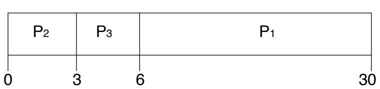
6

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order

P_2 , P_3 , P_1

- The Gantt chart for the schedule is:



- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$
- Much better than previous case
- *Convoy effect* short process behind long process

7

Shortest-Job-First (SJF) Scheduling

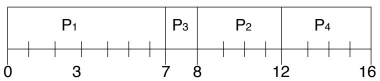
- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time
- Two schemes:
  - nonpreemptive – once CPU given to the process it cannot be preempted until completes its CPU burst
  - preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is known as the Shortest-Remaining-Time-First (SRTF)
- SJF is optimal – gives minimum average waiting time for a given set of processes

8

Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)

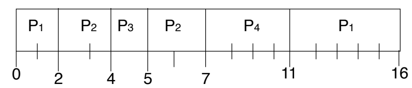


- Average waiting time = (0 + 6 + 3 + 7)/4 = 4

Example of Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (preemptive)



- Average waiting time = (9 + 1 + 0 +2)/4 = 3

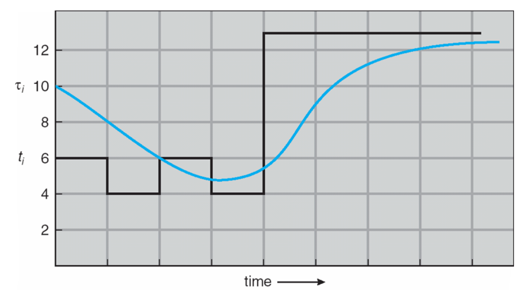
Determining Length of Next CPU Burst

- Can only estimate the length
- Can be done by using the length of previous CPU bursts, using exponential averaging

1. $t_n =$ actual length of n^{th} CPU burst
2. $\tau_{n+1} =$ predicted value for the next CPU burst
3. $\alpha, 0 \leq \alpha \leq 1$
4. Define:

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n$$

Prediction of the Length of the Next CPU Burst



CPU burst (t_i)		6	4	6	4	13	13	13	…
"guess" (τ_i)	10	8	6	6	5	9	11	12	…

Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not matter more
- $\alpha = 1$
  - $\tau_{n+1} = \alpha\, t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\alpha\, t_{n-1} + \ldots + (1 - \alpha)^j \alpha\, t_{n-j} + \ldots + (1 - \alpha)^{n+1} \tau_0$$

- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

13

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer = highest priority)
  - Preemptive
  - nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem: Starvation – low priority processes may never execute
- Solution: Aging – as time progresses increase the effective priority of the process

14

Round Robin (RR)

- Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and re-added to the ready queue.
- If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units at once. No process waits more than $(n-1)q$ time units.
- Performance
  - q large $\Rightarrow$ FIFO
  - q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high

15

Linux Completely Fair Scheduler (CFS)

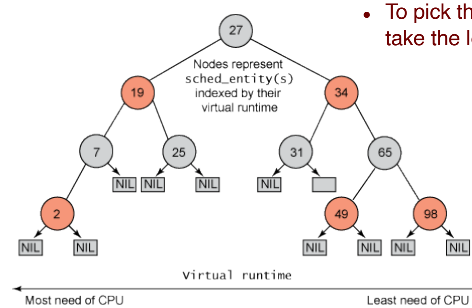
- Incorporated in Linux 2.6.23
  - Removed active and expired arrays
- Uses nanosecond granularity and has no real notion of timeslices
  - /proc/sys/kernel/sched_min_granularity_ns
- No timeslices, no sleep time tracking, no process type identification
- CFS tries to model an “ideal, precise multitasking CPU” – one that could run multiple processes simultaneously, giving each equal processing power

16

Linux CFS

- Measures how much run time each task has had and try to ensure that everyone gets their fair share of time.
- This is held in the vruntime variable for each task, and it is recorded at the nanosecond level. A lower vruntime indicates that the task has had less time to compute, and therefore has more need of the processor
- Maintains a red-black tree ordered by time
  - A virtual timeline of process execution

Linux CFS RB Tree



- The key for each node is the vruntime of the corresponding task.
- To pick the next task to run, take the leftmost node.

Context Switching

- Context switching is at the heart of multiprogramming
  - Stop running process
  - Save the state of the running process
  - Restore saved state of another process
  - Start the new process
- The CPU that is running the process must perform the state management and restarting
  - It doesn't really stop!

Xinu Process Table

- Xinu keeps information about all processes in the *process table*
  - An entry per process, storing process state
  - Called *proctab* with **NPROC** *procent* entries
  - Indexed by the process's PID
- The state of the currently running process is out of date, but for other processes, this is the information used to restore the process at context switch time
  - Not all of the process state – since each process has a distinct stack, the process table only needs to refer to the stack

Items in a Process Table Entry

Field	Purpose
prstate	The current state of the process (e.g., whether the process is currently executing or waiting)
prprio	The scheduling priority of the process
prstkptr	The saved value of the process's stack pointer when the process is not executing
prstkbase	The address of the highest memory location in the memory region used as the process's stack
prstklen	A limit on the maximum size that the process's stack can grow
prname	A name assigned to the process that humans use to identify the process's purpose

21

```
/* process.h –
   provides isbadpid */

/* Maximum number of processes in the system */

#ifndef NPROC
#define NPROC      8
#endif

/* Process state constants */

#define PR_FREE    0   /* process table entry is unused      */
#define PR_CURR    1   /* process is currently running       */
#define PR_READY   2   /* process is on ready queue          */
#define PR_RECV    3   /* process waiting for message        */
#define PR_SLEEP   4   /* process is sleeping                */
#define PR_SUSP    5   /* process is suspended               */
#define PR_WAIT    6   /* process is on semaphore queue      */
#define PR_RECTIM  7   /* process is receiving with timeout  */

/* Miscellaneous process definitions */

#define PNMLEN     16  /* length of process "name"           */
#define NULLPROC   0   /* ID of the null process             */
```

22

```
/* Process initialization constants */

#define INITSTK    65536  /* initial process stack size       */
#define INITPRIO   20     /* initial process priority         */
#define INITRET    userret /* address to which process returns */

/* Reschedule constants for ready */

#define RESCHED_YES 1  /* call to ready should reschedule     */
#define RESCHED_NO  0  /* call to ready should not reschedule */

/*Inline code to check process ID (assumes interrupts are disabled) */

#define isbadpid(x) ( ((pid32)(x) < 0) || \
              ((pid32)(x) >= NPROC) || \
              (proctab[(x)].prstate == PR_FREE))

/* Number of device descriptors a process can have open */

#define NDESC      5    /* must be odd to make procent 4N bytes */
```

23

```
/* Definition of the process table (multiple of 32 bits) */

struct procent {       /* entry in the process table     */
    uint16  prstate;   /* process state: PR_CURR, etc.      */
    pri16   prprio;    /* process priority            */
    char    *prstkptr; /* saved stack pointer         */
    char    *prstkbase;/* base of run time stack      */
    uint32  prstklen;  /* stack length in bytes       */
    char    prname[PNMLEN];    /* process name               */
    uint32  prsem;     /* semaphore on which process waits   */
    pid32   prparent;  /* id of the creating process     */
    umsg32  prmsg;     /* message sent to this process      */
    bool8   prhasmsg;  /* nonzero iff msg is valid       */
    int16   prdesc[NDESC]; /* device descriptors for process */
};

/* Marker for the top of a process stack (used to help detect overflow)
    */
#define STACKMAGIC  0x0A0AAAA9

extern  struct  procent proctab[];
extern  int32   prcount;    /* currently active processes     */
extern  pid32   currpid;    /*currently executing process     */
```

24

Process States

- Each process is assigned a *state*
  - One of the elements of its "state"
- Xinu uses the *prstate* field to record the state
- Symbolic constants defined in *process.h*
- Xinu keeps code and data for processes in memory at all times
  - Larger, general purpose OSes may need additional states for processes e.g., in the system but temporarily moved to secondary storage

Process States

Constant	Meaning
PR_FREE	The entry in the process table is unused (not really a process state)
PR_CURR	The process is currently executing
PR_READY	The process is ready to execute
PR_RECV	The process is waiting for a message
PR_SLEEP	The process is waiting for a timer
PR_SUSP	The process is suspended
PR_WAIT	The process is waiting on a semaphore
PR_RECTIM	The process is waiting for a timer or a message, whichever occurs first

This information is redundant in some cases –
if the process is READY, it is in the ready queue

Scheduling

- Switching processes involves a context switch, and *scheduling*
- The scheduler decides which of the *ready* processes will be run next
  - The scheduler implements the policy, while context switching is the mechanism
- In Xinu, the function *resched* makes the decision
- The basic policy is to choose the highest priority process

Scheduling

- Schedule *round robin* among process with equal priority
  - This means that each of the equal priority processes will be run, one after another
  - Each of them will be run before any of them is run again
- This set of processes from which to choose is all ready processes
  - The currently running process is included

The scheduler

- The scheduler is a function
  - Not an active entity that picks up a process and moves it
- In Xinu, this function is called by a running process to potentially give up the CPU
- To make the scheduler's job easier (and faster) processes are stored in the *ready* list
  - Ordered by priority so the highest priority process is at the head of the list
  - The *ready list* is stored in the queuetab array discussed previously
  - There is a global variable *readylist* that contains the queue ID for this list
  - The currently-running process could be on the ready list, but in Xinu it is not

The scheduler

- The currently-running process relinquishes the CPU by calling the scheduler
  - This process may remain eligible to run and thus the scheduler may change the state from PR_CURR to PR_READY and insert the process into the ready list
- The scheduler doesn't receive an explicit argument to indicate the process's disposition
  - System functions manipulate the *prstate* field and this is inspected by *resched*
- *resched* completes everything except register management
  - Selects new process, removes from ready list, sets currpid, performs other bookkeeping, and calls *ctxsw*


```
/* resched.c - resched */

#include <xinu.h>

/*------------------------------------------------------------------------
 *resched  -  Reschedule processor to highest priority eligible process
 *------------------------------------------------------------------------
 */
void resched(void)        /* assumes interrupts are disabled   */
{
    struct procent *ptold; /* ptr to table entry for old process */
    struct procent *ptnew; /* ptr to table entry for new process */

    /* If rescheduling is deferred, record attempt and return */

    if (Defer.ndefers > 0) {
        Defer.attempt = TRUE;
        return;
    }

    /* Point to process table entry for the current (old) process */

    ptold = &proctab[currpid];
```



```
    if (ptold->prstate == PR_CURR) {  /* process remains running */
        if (ptold->prprio > firstkey(readylist)) {
            return;
        }

        /* Old process will no longer remain current */

        ptold->prstate = PR_READY;
        insert(currpid, readylist, ptold->prprio);
    }

    /* Force context switch to highest priority ready process */

    currpid = dequeue(readylist);
    ptnew = &proctab[currpid];
    ptnew->prstate = PR_CURR;
    preempt = QUANTUM;     /* reset time slice for process   */
    ctxsw(&ptold->prstkptr, &ptnew->prstkptr);

   /* Old process returns here when resumed */

    return;
}
```

Deferring

- *resched* checks Defer.ndefers to determine if scheduling needs to be deferred
- Used when, e.g. a device driver needs to service multiple devices on a single interrupt
- Temporarily disables running the next process
- We will defer this discussion for now